Alexandre Vasconcelos Leite

Suporte à reflexão computacional em ambientes de desenvolvimento visual de software

Florianópolis - SC

2001

UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Alexandre Vasconcelos Leite

Suporte à reflexão computacional em ambientes de desenvolvimento visual de software

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação.

Luiz Carlos Zancanella
(orientador)

Florianópolis, fevereiro de 2001

Suporte à reflexão computacional em ambientes de desenvolvimento visual de software

Alexandre Vasconcelos Leite

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação, Área de Concentração Sistemas de Computação, e aprovada em sua forma final, pelo Programa de Pós-graduação em Ciência da Computação.

Luiz Carlos Zancanella, Dr.
(Orientador)

Fernando A Ostuni Gauthier, Dr.
(Coordenador do Curso)

Banca Examinadora

Luiz Carlos Zancanella, Dr.
(Presidente)

Olinto José Varela Furtado, Dr.

Ricardo Pereira e Silva, Dr.

Rômulo Silva de Oliveira, Dr.

A minha esposa, Marisa, e aos filhos, Lucas e Mariana, pelo incentivo, paciência, colaboração, compreensão, carinho, amor e por fazerem parte da minha vida.

AGRADECIMENTOS

Ao corpo docente do CPGCC, que tive o privilégio de conhecer, em especial ao orientador, professor Luiz Carlos Zancanella, por ter abraçado esta causa e compartilhado seu conhecimento, e ao professor Ricardo Pereira e Silva, pelas valiosas sugestões e atenção;

aos colegas da Divisão de Ensino de Informática (DEInfo) do Colégio Técnico Industrial Prof. Mário Alquati (CTI), Fundação Universidade Federal do Rio Grande (FURG), e também à CAPES pelo suporte à realização deste curso;

finalmente, agradecimentos especiais ao amigo Alessandro Duarte de Moraes, exímio projetista de software, pela ajuda e atenção, e ao professor e amigo Mauro Nicola Póvoas, pela revisão do texto.

RESUMO

Uma evolução da indústria do software, em relação às ferramentas disponíveis para a programação, deu-se com o surgimento dos ambientes de desenvolvimento visual, que combinam recursos textuais (linguagens) e gráficos (ícones e janelas) para a confecção de software. O problema surge quando o ambiente não oferece condições suficientes para que se promovam extensões à linguagem, apresentando-se como uma estrutura fechada de forma que sua utilização seja ao estilo "caixa preta". Como esses ambientes disponibilizam componentes para a confecção de aplicações, normalmente permitem apenas que funcionalidades sejam estendidas em forma de novos componentes.

Este trabalho apresenta uma proposta de suporte à reflexão computacional para os ambientes visuais e o *OP*MOP, como uma implementação desta proposta.

O suporte, voltado à reflexão comportamental, sugere a criação de quatro componentes: um como responsável pelo processo de reificação; outro encarregado de prover o resultado da reificação em componentes visuais; um terceiro para promover todo o suporte de interceptação de mensagens; e um quarto contendo facilidades na definição de quais métodos serão reflexivos.

A implementação *OP*MOP é um conjunto de componentes, desenvolvidos para o ambiente Delphi, que estende recursos reflexivos à linguagem Object Pascal. *OP*MOP é baseado no suporte proposto com as devidas adaptações ao ambiente.

Palavras-chave: reflexão computacional, ambientes de desenvolvimento de software, componentes

ABSTRACT

An evolution of the software industry in relation with programming tools happened with the appearance of visual development environments that combine textual resources (languages) and graphics (icons and windows) to build software. Problems occur when the environment doesn't offer enough conditions to promote language extensions, looking like a closed structure to be used in a black-box style. As these environments offer components to build applications, they usually allow functionalities to be extended only as new components.

This work presents a support proposal to computational reflection for visual environments and *OP*MOP, as an implementation of this proposal.

The support, focused on behavioural reflection, suggests the creation of four components: one responsible for the reification process; another responsible for providing the result of this reification in visual components; a third to promote the whole support message interception; and a fourth one containing means to define which methods will be reflexives.

The implementation *OP*MOP is a group of components, developed for the Delphi environment, extending reflexive resources to the Object Pascal language. *OP*MOP is based on the proposed support with the necessary adaptations to the environment.

Keywords: computational reflection, software development environment, components.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

ABREVIAÇÕES E SIGLAS

CA – Componente Associa

CE – Componente Execução

CR – Componente Reifica

CT – Componente Texto

COM – *Component Object Model*

CORBA – *Commom Object Request Broker Architecture*

DCOM – *Distributed Component Object Model*

DLL – *Dynamic Link Library*

GUI – *Graphical User Interface*

IDL – *Interface Definition Language*

MLI – *Metalevel Interception*

MOP – *Metaobject Protocol*

MVC – *Model-View-Controller*

OMG – *Object Management Group*

OWL – *Object Window Library*

VB – Visual Basic

VCL – *Visual Components Library*

WCOP – *Workshop Component-Oriented Programming*

Capítulo 1
INTRODUÇÃO

A Engenharia de Software possui como premissa a diminuição do esforço na confecção de software, sem esquecer, no entanto, da qualidade e da confiabilidade. Uma vez que os softwares estão cada vez mais complexos, a evolução das ferramentas de desenvolvimento, para suportar tais complexidades, dá-se como uma necessidade. Os paradigmas de programação, por permitirem a representação de abstrações do mundo real, também precisam evoluir. Na indústria do software, destaca-se a introdução do paradigma da orientação a objetos em algumas linguagens como uma das evoluções. Uma outra evolução foi o surgimento dos ambientes de desenvolvimento visual de programação, onde os softwares passaram a ter uma aparência gráfica ao invés de apenas textual.

A reflexão computacional, aplicada ao modelo de objetos, propõe uma nova arquitetura de software [FOO93]. O termo reflexão computacional surgiu em 1982 por Brian Smith [SMI82] no sentido de estender recursos às linguagens de programação. Mais tarde, em 1987, Pattie Maes [MAE87] propôs o uso da reflexão em linguagens baseadas em objetos. Esta nova arquitetura introduz o conceito de níveis de programação. No nível base, estão todas as preocupações referentes ao domínio da aplicação no qual o software está baseado. Qualquer evento que fuja da funcionalidade da aplicação é então desviada (refletida) à outro nível, dito metanível, para o tratamento desta situação.

Conceitualmente, a reflexão computacional é uma técnica que permite a um sistema obter informações sobre ele mesmo e usar estas informações para alterar o comportamento de seus componentes [MAE87]. As informações sobre a estrutura do

nível base são disponibilizadas ao metanível por um processo de reificação[1], passando estas informações a serem chamadas de meta-informações. A funcionalidade do metanível é baseado nos modelos de classe e de objetos. No primeiro, tem-se a reflexão estrutural, que atua na estrutura da classe com operações do tipo: consulta e alteração dos métodos, inclusão de novos métodos, remoção de métodos existentes, consulta às classes ascendentes, dentre outras. No segundo modelo, tem-se a reflexão comportamental, que atua na execução do objeto do nível base através da interceptação de mensagens, sem alterar sua estrutura.

As funcionalidades da reflexão computacional são providas pelo MOP (*Metaobject Protocol*). Um MOP é uma interface que permite mudanças incrementais nos recursos de linguagens de programação. Então, uma aplicação que utiliza um MOP terá, à sua disposição, o modelo de reflexão conforme a implementação do MOP.

Algumas implementações de MOP estendem recursos de reflexão computacional tanto às linguagens que possuem algum suporte, tais como Java e Smalltalk, quanto àquelas sem qualquer suporte a reflexão, como o C++. Numa outra situação, novas linguagens de programação são propostas já com a incorporação de um MOP específico.

Analisando-se a evolução dos ambientes de programação junto aos recursos que o usuário dispõe para o desenvolvimento de aplicações, observa-se que as ferramentas levam algum tempo para disponibilizar novas funcionalidades, ainda que, às vezes, estas sejam baseadas em conceitos sedimentados. A orientação a objetos é um caso típico. Excetuando-se Smalltalk, Java e Eiffel, que já surgiram com os conceitos da OO, muitas outras linguagens, como C e Pascal, passaram a suportar a OO depois de estarem conhecidas. A reflexão computacional também é mais um caso. Novamente à exceção de Smalltalk e Java, as linguagens C e Pascal, mesmo em ferramentas de programação mais recentes como os ambientes de desenvolvimento visual, nada possuem de suporte à reflexão.

---

[1] Traduzido do termo em inglês *reification*

Os ambientes de desenvolvimento visual são ferramentas híbridas que possuem, além de uma linguagem de programação (geralmente OO), recursos textuais e gráficos para a confecção de software. Entre outras características, estes ambientes facilitam o desenvolvimento de aplicações pela utilização de recursos gráficos (ícones e janelas). Em algumas ferramentas disponíveis, tais como, Visual C++, Visual Basic, VisualWorks, Delphi e JBuilder, observa-se que a reflexão computacional não é explorada aproveitando as características de seus ambientes. Mesmo naquelas linguagens consideradas reflexivas, como Java e Smalltalk, os ambientes não oferecem facilidades visuais ao desenvolvedor para que construam aplicações reflexivas.

Qualquer proposta de estender funcionalidades a um ambiente visual de programação deve abordar tanto a linguagem de programação quanto as questões visuais, uma vez que estes dois elementos estão juntos. O problema surge quando o ambiente não oferece condições suficientes para que se promovam extensões às linguagens, apresentando-se como uma estrutura fechada de forma que sua utilização seja ao estilo "caixa preta". Pelo fato desses ambientes disponibilizarem componentes para a confecção de aplicações, normalmente permitem apenas que funcionalidades sejam estendidas por intermédio de novos componentes.

Neste sentido, apresenta-se uma proposta de suporte à reflexão computacional em ambientes de desenvolvimento visual de software baseada na criação de quatro componentes, voltados especificamente à reflexão comportamental. São eles: Componente Reifica (CR), responsável pelo processo de reificação; Componente Texto (CT), encarregado de prover o resultado da reificação em componentes visuais; Componente Associa (CA), para promover todo o suporte de interceptação de mensagens; Componente Execução (CE) contendo facilidades na definição de quais métodos serão flexivos.

Apresenta-se, também, uma implementação do suporte proposto, denominada ***OP*MOP**, como um conjunto de componentes para o ambiente Delphi que estende funcionalidades à linguagem Object Pascal. ***OP*MOP** contém os quatro componentes

propostos implementados com os devidos ajustes para uma perfeita integração com o ambiente Delphi.

1.1. Organização do trabalho

O presente trabalho está organizado em seis capítulos e quatro apêndices. Os três primeiros capítulos apresentam a introdução e a fundamentação teórica, enquanto que os restantes, a proposta de suporte e uma implementação.

O capítulo 2 comenta os conceitos básicos da reflexão computacional, passando pelos sistemas reflexivos e apresentando os modelos de reflexão aplicados ao modelo de objetos. Traz também as metas de um MOP.

No capítulo 3, as características dos ambientes de desenvolvimento visual de software são discutidas, apresentando-as pela comparação entre três ferramentas atualmente existentes.

O capítulo 4 apresenta uma proposta de suporte à reflexão computacional em forma de componentes.

Já o capítulo 5 traz uma implementação do suporte proposto, denominado *OP*MOP, desenvolvido para o ambiente Delphi.

O capítulo 6 apresenta as considerações finais do trabalho, onde algumas perspectivas de continuidade são propostas.

No apêndice A é apresentado uma visão do Delphi, destacando apenas as funcionalidades que serão úteis a implementação *OP*MOP.

O apêndice B apresenta, em linhas gerais, a abordagem dos *frameworks* e componentes como uma alternativa à reutilização de software proposta pela Engenharia de Software.

A preocupação com o desempenho do processo de interceptação de mensagens de *OP*MOP é apresentada através da comparação dos resultados de dois testes, no apêndice C.

E, finalmente, no apêndice D, os programas fontes utilizados como exemplos no uso dos componentes *OP*MOP são mostrados.

Capítulo 2
REFLEXÃO COMPUTACIONAL

A Engenharia de Software evolui no sentido de buscar a confecção de software a um custo baixo, de qualidade e confiável. Não importando a técnica, a metodologia ou a ferramenta de desenvolvimento adotada, as preocupações ainda são as mesmas. Isto porque o software está cada vez mais complexo e, para torná-lo viável em termos de prazos, exige-se uma maior participação do número de pessoas envolvidas no seu desenvolvimento. Baseado nisto, há um esforço das formas de desenvolvimento em direção a minimizar o trabalho manual, como por exemplo o uso de programas geradores de código. Sempre que possível os programas devem ser gerados automaticamente a partir do projeto, todavia ainda faz-se necessária a participação manual na codificação, uma vez que as linguagens de programação possuem baixa expressividade.

Na etapa de programação, as principais preocupações são a codificação e a integração progressiva dos módulos do software, até se chegar ao sistema físico final. É claro que nesta etapa a busca pela qualidade deve ser constante, assim como em todo o processo de desenvolvimento. Desta forma, a capacidade dos programadores em implementar um sistema de alta qualidade e livre de erros depende não só da natureza do projeto criado, como também da linguagem de programação adotada como ferramenta de desenvolvimento [KIP93].

É possível agrupar as diferentes linguagens de acordo com seus paradigmas de programação. Destes grupos, o enfoque neste trabalho é o do paradigma da orientação a objetos. Além de estarem presentes em muitas ferramentas de desenvolvimento de

software, a orientação a objetos possui várias vantagens de uso, tais como reutilização de código e ocultamento de detalhes do usuário, entre outras.

Brian Foote [FOO93] coloca que "a sinergia da reflexão com a programação e as técnicas de projeto orientados a objetos traz promessas de mudanças dramáticas na forma como nós pensamos, organizamos, implementamos e usamos as linguagens de programação". É neste sentido, então, que a reflexão computacional no modelo de objetos apresenta-se como uma nova arquitetura de software.

2.1. Conceituação

Brian Smith [SMI82] propôs a reflexão computacional em 1982 como uma extensão às linguagens de programação. Pattie Maes [MAE87], em seu trabalho de doutoramento "*Computational Reflection*", sugere o uso da reflexão no modelo de objetos.

Pode-se analisar a expressão "reflexão" sob dois aspectos: como o ato ou efeito de meditar, pensar, raciocinar e ponderar e como uma propriedade física de modificação da direção de propagação da luz, calor e som, desviando da primeira direção. Como ato de meditação, a reflexão computacional pode ser caracterizada pela capacidade dada a um sistema de "pensar" a respeito dele mesmo, atuando sobre si mesmo e não sobre o que o sistema deva produzir, realizando deduções e computações sobre seus dados internos. Como propriedade física, este significado relaciona-se mais diretamente com a forma de implementação da reflexão computacional no modelo orientado a objetos: ao ser enviada uma mensagem a um objeto, ela é desviada ao seu meta-objeto correspondente, no qual é realizada a reflexão, e, como resultado desta reflexão, passa a realizar computações no metanível.

Conceitualmente, **reflexão computacional** é o comportamento exibido por um sistema computacional que atua em algum domínio e que incorpora estruturas representando ele mesmo. Estas estruturas permitem ao sistema responder questões e suportar ações sobre ele mesmo [MAE87].

Lisboa [LIS97] define reflexão computacional como toda atividade que um sistema computacional realiza sobre si mesmo, de forma separada das computações em curso, com o objetivo de resolver seus próprios problemas e obter informações sobre suas computações.

Na visão de Steel [STE94], reflexão computacional é a capacidade de um sistema computacional interromper o processo de execução (por exemplo, quando ocorre um erro), realizar computações ou fazer deduções no metanível e retornar ao nível de execução, traduzindo o impacto das decisões para então retomar o processo de execução.

Maes [MAE88] destaca ainda alguns tipos de aplicações onde seria interessante o uso da reflexão computacional, tais como estatísticas de desempenho de sistemas, ferramentas de depuração e programas de monitoramento, entre outros.

2.2. Sistema reflexivo

No modelo de objetos, um sistema é dito reflexivo quando provê mecanismos para suporte à reflexão computacional. Nele podem ser identificadas duas partes distintas: um metanível, também conhecido como nível de gerenciamento, e um nível base (ou nível da aplicação). No **metanível** encontram-se as informações (as meta-informações) que representam a estrutura e comportamento do nível base, enquanto que no **nível base** estão os requisitos funcionais do sistema baseados no domínio da aplicação. No nível base o programador envolve-se apenas com a solução do problema em relação ao domínio na qual sua aplicação está inserida. Já no metanível ficam os requisitos não funcionais e estes são independentes do tipo de aplicação, como, por exemplo, aspectos de segurança do sistema. Esta clara separação entre as funcionalidades básicas não apenas contribui para resolver problemas sobre um determinado domínio, mas também para a organização interna do sistema [MAE88].

A reflexão computacional dá-se pelas etapas de reificação e de reflexão propriamente dita. A **reificação** disponibiliza ao metanível as informações do nível da

aplicação. Neste processo, as atividades computacionais de uma classe no nível base são transformadas em dados para o metanível, as quais podem ser manipuladas pelo metanível. Após feito isto, estes dados são transformados em atividades computacionais e o fluxo do controle retorna do metanível para o nível base. Esta última operação é chamada de **reflexão** [NAK92]. A figura 2.1 mostra um esquema simples de um sistema reflexivo em que aparecem as operações de reificação e reflexão juntamente com o MOP[2].

Uma questão importante em relação a sistemas reflexivos é decidir que tipo de informação será disponibilizada no metanível. Tais informações incluem a classe do objeto, a qual fornece acesso ao nome da classe e seu tipo; a herança, com informações sobre as classes ascendentes; e a estrutura do objeto, disponibilizando informações sobre seus métodos (assinatura, código e acesso).

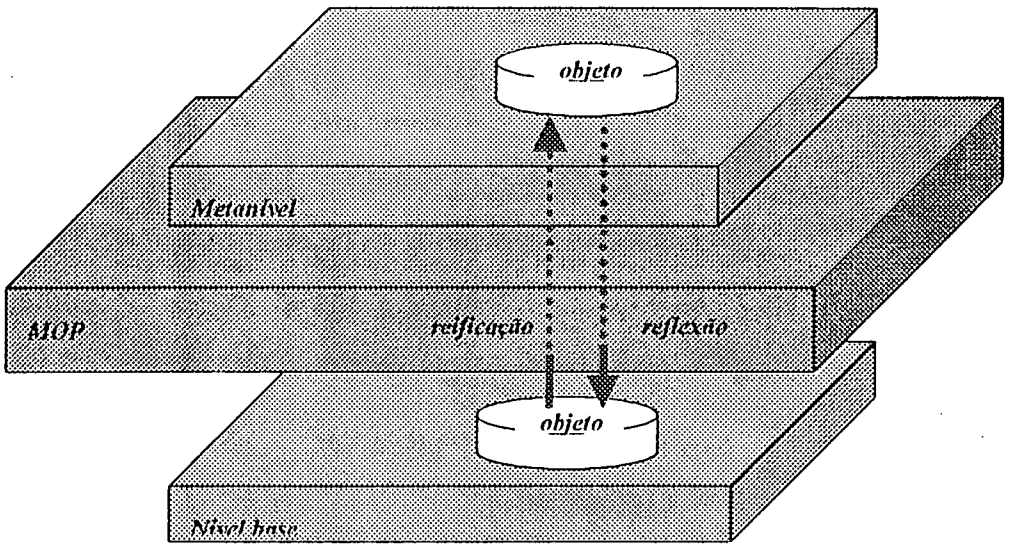


FIGURA 2.1: SISTEMA REFLEXIVO

[2] Detalhado mais adiante, neste mesmo capítulo

2.3 Metaclasse e meta-objeto

Em linguagens de programação orientadas a objetos, a reflexão dá-se de duas formas: ou por metaclasse ou por meta-objeto. A reflexão por **metaclasse** faz com que o metanível contenha informações sobre a estrutura do nível base, tais como atributos e assinaturas de comportamentos, classes ascendentes e descendentes, encapsulamento, comportamentos dinâmicos e estáticos e outras. Uma vez que a estrutura da classe nível base pode ser alterada, todas suas instâncias a partir desta alteração também serão. Uma metaclasse é uma classe que descreve a estrutura de uma outra classe.

Dependendo da linguagem de programação, classes e metaclasses podem ser representadas por objetos. Isto permite que as metaclasses possuam capacidades comportamentais para a instanciação dinâmica de classes, da mesma forma que as classes permitem a instanciação de objetos.

A metaclasse independe do código do domínio escrito no nível de aplicação, podendo, então, ser reutilizada e agrupada. Uma implementação que provê esta característica é o Guaraná [OLI98a].

Aparece ainda, neste mesmo espaço do metanível, o **meta-objeto,** como um objeto que contém detalhes do comportamento do objeto nível base. Foote [FOO93] coloca que um meta-objeto define, implementa, dá suporte ou participa de alguma maneira da execução da aplicação. Nesta instância, o meta-objeto contém alguns comportamentos (não necessariamente todos) do objeto nível base associado. Isto faz com que a reflexão seja restrita ao método chamado, tornando-se uma reflexão computacional particular de cada método e não de todo o objeto. Surge daí, então, dois grupos de métodos de um objeto reflexivo: os reflexivos e os não reflexivos [LIS97].

2.3.1. Reflexão estrutural e comportamental

As duas questões relacionadas ao metanível, a metaclasse e o meta-objeto, dão origem a dois tipos de reflexão. Diz-se que uma reflexão é **estrutural** quando se dá sobre a classe do nível base e ocorre tanto em tempo de compilação quanto em execução e é dita **comportamental** quando utiliza o modelo de meta-objetos e ocorre somente em tempo de execução.

No caso da Reflexão Estrutural, o metanível é composto por metaclasses, as quais contém informações sobre os aspectos estruturais do nível base, como descrição dos atributos e métodos que irão compor suas instâncias. Se esta estrutura pode ser alterada, então a estrutura e comportamento de suas instâncias também serão. A reflexão estrutural permite não só obter informações sobre o objeto do nível de aplicação como também realizar transformações sobre a estrutura estática da classe, modificando seus atributos e comportamentos. As regras de encapsulamento existentes numa classe, que asseguram o ocultamento de atributos e comportamentos, podem ser violadas, uma vez que a metaclasse contém toda a sua estrutura e pode, inclusive, alterá-la.

Na Reflexão Comportamental, o metanível é composto por meta-objetos, os quais contém informações sobre os aspectos de comportamento dos objetos do nível base, sem alterar sua estrutura. O modelo de meta-objetos difencia-se do modelo de metaclasses, principalmente, por sua associação por objetos e não por classes.

A associação entre os dois objetos em um sistema reflexivo é importante porque todas as mensagens enviadas pela aplicação ao objeto do nível base da associação (chamado de objeto reflexivo) são desviadas para o meta-objeto, que detém o controle da execução. Uma vez que o meta-objeto recebe uma mensagem reificada através de um mecanismo de interceptação, ele deve ser capaz de tratá-la e, após realizar a reflexão computacional, voltar ao objeto reflexivo.

A figura 2.2, adaptada de Campo [CAM97], mostra o fluxo de controle entre objetos e meta-objetos. Nela percebe-se que uma mensagem pode ser interceptada

quando ela é enviada ou quando ela é recebida pelo destinatário. Independente da forma de interceptação, o protocolo de meta-objetos deve fornecer mecanismos para recuperar a informação relativa ao destinatário da mensagem, o método a ser ativado e os argumentos da mensagem. A reificação destes componentes possibilita que o meta-objeto reenvie a mensagem ao objeto da aplicação para a execução do método original [CAM97].

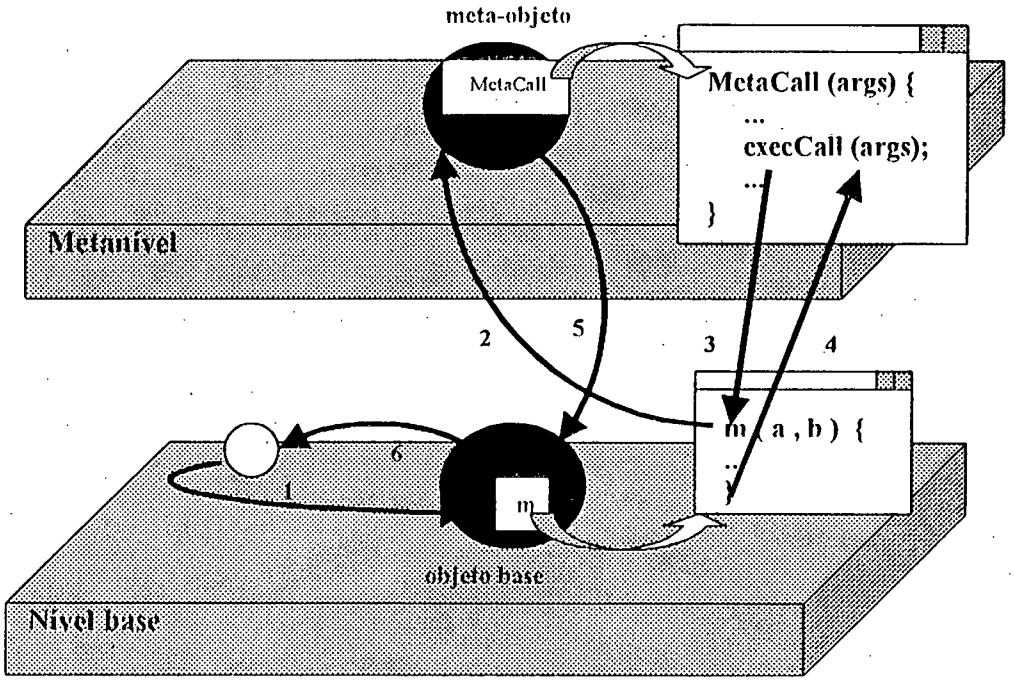


FIGURA 2.2: FLUXO DE CONTROLE ENTRE OBJETOS E META-OBJETOS

Na etapa **(1)** uma mensagem **m** é enviada a um objeto da aplicação. Esta mensagem, então, é desviada (refletida) para o meta-objeto associado **(2)**. O meta-objeto, após realizar alguma computação, decide executar o método **m** original, presente no nível base **(3)**. Após a execução do método, o fluxo retorna para o meta-objeto **(4)**. O meta-objeto, por sua vez, retorna o controle ao nível base após executar alguma função adicional **(5)** e, finalmente, ao objeto que enviou a mensagem **m (6)**.

2.4. *Metaobject Protocol (MOP)*

"Um MOP é um mecanismo usado para o controle do comportamento de uma aplicação e para implementar políticas não funcionais tais como tolerância a falhas e execução distribuída" [ROB99]. Com este comentário, Bert Robben apresenta as duas principais funções do MOP. Um **MOP** pode ser descrito como uma interface que possibilita aos objetos do nível base comunicarem-se com os objetos do metanível Tecnicamente falando, um MOP é um *framework*[3]. Assim, é no MOP que deve ser implementado o mecanismo de interceptação de mensagens que transferirá o controle da execução do nível base para o metanível. Segundo Zimmermann [ZIM96], a interceptação do metanível (*metalevel interception - MLI*) é a transferência do controle do nível base para o metanível e pode ser implementada tanto em linguagens interpretadas quanto compiladas. Na implementação do OpenC++ [CHI95], por exemplo, um pré-processador envolve o comportamento original que invoca o objeto de controle na metaclasse no início e no final da execução.

A implementação de uma MLI deve ser feita de uma forma transparente, ou seja, a classe do nível base que tenha a chamada de um método interceptada não deve perceber que houve um desvio do controle para o metanível. Lembrando que uma MLI tem uma relação direta com o comportamento em tempo de execução do sistema, sua implementação deve ser tal que produza uma sobrecarga mínima.

Na provisão de um suporte de meta-objetos surgem, basicamente, três aspectos que devem ser contemplados pela implementação da linguagem:

(a) transformação da informação do programa necessária para realizar seu tratamento no metanível;

[3] Modelos formam a base conceitual de diferentes arquiteturas de software. Mary Shaw [SHA95] classifica um MOP como um modelo de *framework* que atende a classes de problemas e domínios específicos. Embora sendo a reutilização de código e projeto a principal motivação de *frameworks*, a associação de MOP como um *framework* está diretamente relacionada com a propriedade do MOP em definir sistemas abertos, flexíveis, e extensíveis. O anexo 2 traz a abordagem de *frameworks* orientados a objetos.

(b) associação entre os objetos do nível base com seus correspondentes meta-objetos;
(c) a ativação desses meta-objetos quando um objeto do nível base deve realizar alguma operação controlada por um meta-objeto.

A implementação particular destes aspectos fornecida por uma linguagem reflexiva constitui o MOP dessa linguagem. A operação de reificação, já comentada, surge no primeiro aspecto. Os demais referem-se aos mecanismos que definem como a associação é implementada e como se inicia o processo de reflexão. Existem, basicamente, duas possibilidades para implementação de computações reflexivas [MAE88]: a responsabilidade pode ser atribuída ao objeto de nível base, que neste caso contém código mencionando seu meta-objeto, ou pode ser realizada automaticamente pelo sistema de suporte de execução da linguagem.

2.4.1. Metas num projeto de MOP

Um MOP define um protocolo padrão para a associação de meta-objetos com objetos do nível base e a ativação dos meta-objetos em resposta a eventos produzidos no nível base. A complexidade da implementação dos mecanismos necessários para prover a capacidade de reflexão depende da linguagem de programação. Smalltalk provê várias facilidades para implementar computações reflexivas [JOH89], enquanto que no C++, por ser uma linguagem estática e fechada, a implementação de um suporte básico torna-se mais complexa.

No projeto de um MOP, é importante que estejam contempladas algumas características fundamentais, como robustez, abstração e extensibilidade. Ele é robusto quando permite ao programador depender da funcionalidade documentada das classes do protocolo. Em outras palavras, a integridade semântica deve ser reforçada e o sistema, quando sofrer adaptações, não deve apresentar efeitos colaterais, nem nele nem nos demais sistemas. A característica da abstração surge no momento em que detalhes de implementação são ocultados do programador. Dependendo do nível exigido de reflexão sobre o protocolo, a revelação de certos detalhes, em alguns casos, poderá facilitar a compreensão da funcionalidade de um módulo. Uma delicada questão é saber

o ponto exato entre a abstração e a revelação [BRE92]. O projeto de um MOP é extensível quando o protocolo possui a capacidade de sofrer o acréscimo de novas funcionalidades sem afetar as demais metas.

Kiczales [KIC91] afirma que, tradicionalmente, os usuários lidam nas linguagens de programação com abstrações tipo "caixa preta" e que uma linguagem que incorpora a idéia do MOP as deixam ao estilo "caixa branca". Esta característica de torná-la como uma linguagem aberta permite aos usuários das linguagens com MOP ajustarem-na de acordo com suas necessidades particulares. Afirma, ainda, que um MOP deve ser utilizado em projetos de software complexos, não sendo aconselhável, portanto, seu uso em pequenos sistemas, sob pena de afetar o desempenho deste sistema.

Note-se, entretanto, que algumas metas são conflitantes entre si. Se o protocolo exige que sua robustez deva ser priorizada como meta de projeto, sua capacidade de ser extensível pode ficar prejudicada, a fim de minimizar os riscos de problemas futuros. Sendo o protocolo um software, pode-se imaginar também um conflito em relação à facilidade de uso e eficiência. Uma vez que a eficiência é sentida em tempo de compilação, em tempo de carga e em tempo de execução, ela deve ser enfatizada no momento apropriado.

Observando-se estas metas, percebe-se o quanto é difícil um protocolo contemplar todas ao mesmo tempo. Muitas vezes as implementações preocupam-se na otimização de algumas em detrimento das demais. Cabe ao projetista do MOP, então, a tarefa de escolher de que forma estas metas serão implementadas. Suas decisões serão baseadas, também, no ambiente na qual a linguagem de programação está inserida. O capítulo seguinte comenta os ambientes de desenvolvimento visual de programação e a forma como o protocolo pode tirar proveito deles.

2.5. Trabalhos existentes

Foram encontradas várias implementações de reflexão computacional. Destas, algumas foram selecionadas para a apresentação, segundo os critérios abaixo:

(a) serem linguagens de programação orientadas a objetos;

(b) estenderem recursos às linguagens OO que não possuem os conceitos de reflexão ou possuem poucas funcionalidades;

(c) implementarem o mecanismo de interceptação de mensagens de maneiras diferentes.

2.5.1. OpenC++

Shigeru Chiba, do Institute of Information Science and Eletronics, University of Tsukuba, apresenta várias contribuições utilizando C++ e Java. Uma delas, o **OpenC++** [CHI95, CHI96], mostra uma implementação baseada na linguagem C++ que lida diretamente com o código fonte dos programas nível base e metanível. A figura 2.3 mostra os passos executados para a criação de um programa reflexivo.

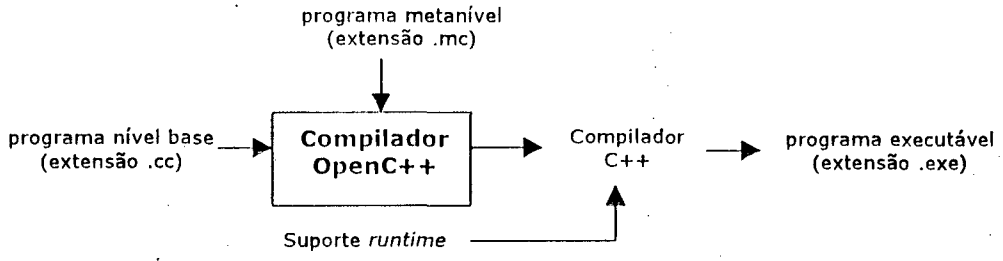


FIGURA 2.3: ESQUEMA DO OPENC++

Após o usuário ter escrito as classes nível base e metanível, o compilador OpenC++ fará a tradução destes códigos em C++ para que o compilador C++, então, gere o código executável. Se nenhuma metaclasse está associada à nível base, o OpenC++ é idêntico ao C++ e esta associação é de "um para um", ou seja, uma metaclasse é única para a nível base e a classe nível base possui apenas uma metaclasse.

Exemplo: supor as classes `Person`, do nível base, e `VerboseClass`, do metanível, conforme figura 2.4. Para a associação do programa nível base com o programa do

metanível, o OpenC++ versão 2.5 introduz um novo tipo de declaração em C++, chamada de **metaclass** (destacado em Person.cc)

```
// Person.cc
// ----------------------------------------
# include <stdio.h>

metaclass VerboseClass Person;

class Person {
      public:
            Person (int i);          { age = i; }
            int Age();               { return age; }
            int BirthdayComes();     { return ++age; }
      private:
            int age;
};

main() {
      Person luis(24);
      printf("age %d\n", luis.Age());
      printf("age %d\n", luis.Birthday());
}
```

```
// VerboseClass.mc
// ----------------------------------------
# include "mop.h"

class VerboseClass : public Class {
      public:
            Ptree* TranslateMemberCall (Environment*, Ptree*, Ptree*,
                                        Ptree*, Ptree*);
};

Ptree* VerboseClass::TranslateMemberCall (Environment* env, Ptree* object,
                                          Ptree* op, Ptree* member,
                                          Ptree* arglist)
{
      return Ptree::Make("(puts(\"%p()\"), %p)", member,
                         Class::TranslateMemberCall(env, object, op, member,
                                                    arglist));
}
```

FIGURA 2.4: EXEMPLO DO OPENC++

Esta declaração define a metaclasse (`VerboseClass`) para uma classe (`Person`) e deve aparecer antes da definição de `Person`. Aqui `VerboseClass` é uma subclasse de **Class**. Class é uma metaclasse padrão que provê uma interface para acesso da definição da classe. Para alterar seu comportamento, o programador define uma subclasse de Class.

O MOP (*Metaobject Protocol*) é o responsável pela tradução do fonte OpenC++ em C++, e não apenas representa os aspectos estruturais dos meta-objetos mas como também permite ao programador alterar o comportamento do programa [CHI97]. O mecanismo de interceptação de mensagens é baseado num *wrapper*, envolvendo o comportamento original, que invoca o objeto de controle na metaclasse no início e no final da execução.

2.5.2. OpenJava

Uma outra implementação apresentada por Chiba é o **OpenJava** [CHI98a]. Ela é baseada na linguagem Java e possui o modelo muito semelhante ao OpenC++, conforme pode ser observado na figura 2.5.

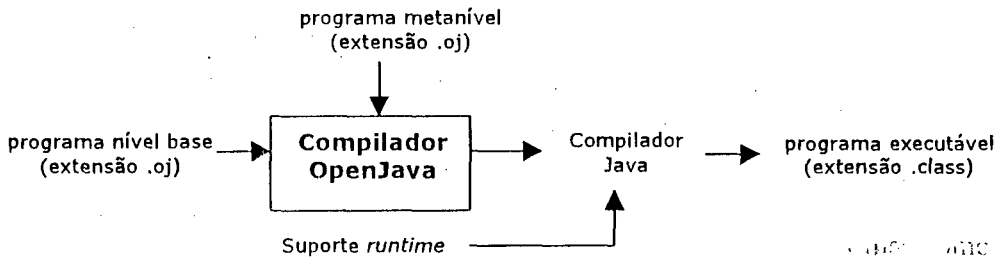


FIGURA 2.5: ESQUEMA DO OPENJAVA

O compilador OpenJava consiste em três etapas: pré-processamento, tradução de fonte-para-fonte de OpenJava para Java e a compilação. O MOP do OpenJava é uma interface para controlar o tradutor na segunda etapa. Ele permite especificar como uma característica estendida do OpenJava é traduzida em código Java.

Uma característica estendida do OpenJava é fornecida como um software adicional para o compilador. Esse software consiste não só de um programa metanível mas como também de um código de suporte em tempo de execução. Esse código provê classes usadas pelo programa do nível base traduzido em Java. O programa do nível

base, escrito em OpenJava, é traduzido para Java conforme o programa do metanível e é dinamicamente ligado com o código suporte em tempo de execução.

O MOP do OpenJava segue três passos:

- decide como o programa do nível base deveria parecer;
- compreende o que deveria ser traduzido e que código suporte é necessário em tempo de execução;
- escreve um programa metanível para executar a tradução e também escreve o código suporte em tempo de execução.

Exemplo: supor as classes `Person`, do nível base, e `VerboseClass`, do metanível, conforme figura 2.6. No arquivo fonte `Person.oj` observa-se, em destaque, **`instantiates VerboseClass`**. Esta notação especial criada pelo OpenJava significa que a semântica da classe `Person` é especificada para ser estendida pela classe `VerboseClass`. Na prática, o código fonte de `Person` é traduzido pelo objeto de `VerboseClass`.

Observa-se, também, no arquivo fonte `VerboseClass.oj`, que a classe do metanível `VerboseClass` é uma classe nível base do ponto de vista da metaprogramação e, de fato, ela declara suas classes do metanível através de **openjava.mop.Metaclass**, embora deveria ser escrita em Java. Ela herda de **openjava.mop.OJClass** e sobrepõe funções membros.

Diferentemente do OpenC++, o MOP do OpenJava não está preparado para que o comportamento de um programa seja alterado em tempo de execução. Oferece apenas o suporte a reflexão em tempo de compilação. As mensagens são interceptadas da mesma forma que o OpenC++, isto é, por intermédio de *wrappers*.

```
// Person.oj
// ----------------------------------------

public class Person instantiates VerboseClass {
      int age;
      publicPerson (int i);          { age = i; }
      public int Age();              { return age; }
      public int BirthdayComes();    { return ++age; }

      public static void main (String[] argv) {
            Person luis = new Person(24);
            System.out.println("age = " + luis.Age());
            System.out.println("age = " + luis.BirthdayComes());
      }
}
```

```
// VerboseClass.oj
// ----------------------------------------
import openjava.mop.*;
import openjava.ptree.*;
import openjava.syntax.*;

public class VerboseClass instantiates Metaclass extends OJClass
{
      public void translateDefinition() throws MOPException {
            OJMethod[] methods = getDeclaredMethods();
            for (int i = 0; i < methods.length; i++) {
                  Statement printer = makeStatement
                        ("java.lang.System.out.println(\"" +
                        methods[i].toString() + " for chamado\");");
                  methods[i].getBody().insertElementAt(printer, 0);
            }
      }
}
```

FIGURA 2.6: EXEMPLO DO OPENJAVA

2.5.3. Javassist

Também baseado em Java surge o **Javassist** (JAVA programming ASSISTant) como uma biblioteca de classes para edição dos *bytecodes*. Permite a definição de novas classes em tempo de execução e a modificação de classes quando a máquina virtual do Java (JVM) é carregada [CHI98b]. Diferentemente do OpenC++ e do OpenJava, o Javassist não é um sistema reflexivo em tempo de compilação e não faz uma tradução fonte-a-fonte. Se for necessário inspecionar uma classe, é feita uma consulta à própria API do Javassist (figura 2.7).

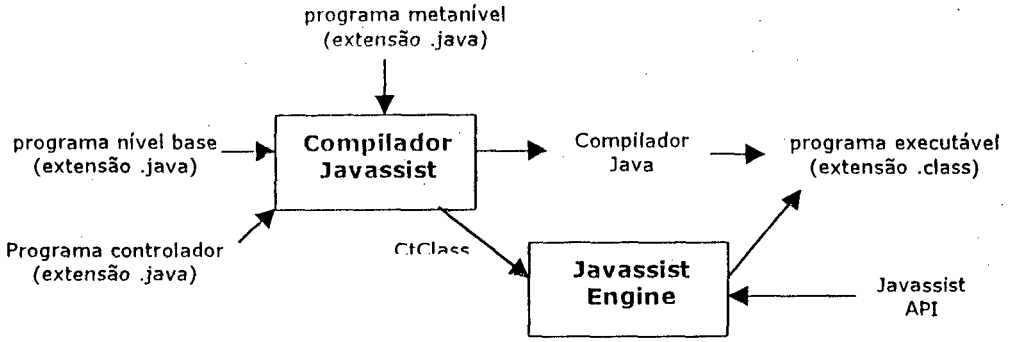


FIGURA 2.7: ESQUEMA DO JAVASSIST

Neste modelo, surge a figura de uma classe controladora que faz a associação entre o programa nível base e o metanível. O compilador gera, então, um objeto *CtClass* (*compile time class*) e o compila em tempo de execução. Neste ambiente, a execução é feita por um *Loader* juntamente com a classe controladora. Para execução sem reflexão, basta invocar apenas a classe nível base, que ficou inalterada. A interceptação de mensagens é feita por *Loader*, pois todas as chamadas aos comportamentos passam por ela.

Exemplo: supor as classes `Person`, do nível base, `VerboseClass`, do metanível e `Main`, como sendo a controladora de `Person` e `VerboseClass`, conforme figura 2.8. No arquivo fonte `Main.java` observa-se, na linha em destaque, a implementação da associação do programa nível base com o do metanível. A função da classe `Main` é, além de definir a associação, controlar a execução da classe base `Person`. Uma outra característica interessante do Javassist reside no fato das classes `Person` e `VerboseClass` não terem nenhuma referência explícita, em seus respectivos códigos fontes, sobre quem é a classe base e quem é a metaclasse. Isto, de certa forma, oferece ao programador uma flexibilidade de execução, pois permite que `Person` seja executada sem qualquer recurso reflexivo (pela linha de comando *$ java Person*), sem que haja a necessidade de alteração em seu código (para a execução de `Person` com recursos reflexivos, neste exemplo, basta usar a linha de comando *$ java javassist.Loader Main*).

```
// Person.java
// ---------------------------------------------
import javassist.reflect.Metalevel;
import javassist.reflect.Metaobject;

public class Person {
        public int age;

        publicPerson (int i);              { age = i; }
        public int Age();                  { return age; }
        public int BirthdayComes();        { return ++age; }

        public static void main (String[] argv) {
                Person luis = new Person(24);
                System.out.println("age = " + luis.Age());
                System.out.println("age = " + luis.BirthdayComes());
        }
}
```

```
// VerboseClass.java
// ---------------------------------------------
import javassist.*;
import openjava.reflect.*;

public class VerboseClass extends Metaobject {
        public VerboseClass (Object self, Object[] a) { super(self, a); }

        public Object trapMethodcall(int id, Object[] args) {
           System.out.println("Metodo "+getMethodName(id) + " foi chamado");
           return super.trapMethodcall(id, args);
        }
}
```

```
// Main.java
// ---------------------------------------------
import javassist.Loader;
import openjava.reflect.ClassMetaobject;
import openjava.reflect.ReflectLoader;

public class Main {

        public static void main (String[] args) throws Throwable {

                Loader cl = (Loader)Main.class.getClassLoader();
                ReflectLoader loader = new ReflectLoader();
                loader.makeReflective("Person", VerboseClass,
                                      ClassMetaobject.class);
                cl.getClassPath().addPath(loader);
                cl.run("Person", args);
        }
}
```

FIGURA 2.8: EXEMPLO DO JAVASSIST

2.5.4. Guaraná

Uma outra importante contribuição estudada veio do Laboratório de Sistemas Distribuídos, Instituto de Computação, Universidade Estadual de Campinas, São Paulo, com a implementação do **Guaraná** [OLI98b]. É uma arquitetura reflexiva baseada no Java que visa simplicidade, flexibilidade, segurança e reutilização de código metanível.

O MOP do Guaraná permite que múltiplos meta-objetos sejam combinados para definir um comportamento de metanível para um objeto, os quais terão reutilização de código [OLI98a]. Entende-se por composição como a capacidade de juntar código para trabalharem em conjunto, dispostos de uma forma hierárquica. A característica de segurança é garantida pelo encapsulamento dos meta-objetos num mesmo *composer*. Este encapsulamento promove o ocultamento dos metaníveis em relação ao nível base e também a capacidade de limitar as habilidades dos meta-objetos.

O modelo adotado para a implementação do Guaraná foi a de modificar o Kaffe OpenVM, uma implementação livre da máquina virtual do Java de autoria de Tim Wilkinson, CEO da Transvirtual Technologies Inc. Esta opção deu-se, segundo os autores, em função do fraco desempenho das alterações dinâmicas e das limitações reflexivas que os modelos anteriores apresentam (modificações de *bytecodes* pelos tradutores, *ClassLoader* para a execução e a inserção de código para o tratamento da interceptação das mensagens). Esta modificação na máquina virtual é que possibilita o tratamento da interceptação das mensagens.

2.5.5. Outras implementações

Foram encontradas outras implementações e, embora não menos importantes, não foram detalhadas neste trabalho. Eis algumas:

ALBEDO: Apresenta uma infra-estrutura de meta-objeto para Smalltalk [BEK93]

CLOS (Common Lisp Object System). Define um protocolo reflexivo para a manipulação de construções da linguagem Lisp [BOB88, KIC91]

Kava: é o nome da nova versão do Dalang, uma extensão do Java que implementa a reflexão através de classes *wrappers*. Kava pretende aplicar a reflexão de uma forma transparente quando existir o uso de componentes de software COTS (Commercial Off-the-Shelf) [WEL98, WEL99]

Oberon-2. Apresentado por Mössemböck [MOS99] como uma linguagem de programação semelhante ao Pascal e Modula-2 que provê um mecanismo de reflexão onde a meta-informação não é obtida por metaclasse, mas sim por uma estrutura enumerada armazenada em disco.

RbCl (Reflection Based Concurrent Language). É uma linguagem concorrente orientada a objetos com arquitetura reflexiva para ambientes distribuídos, proposta por Ichisugi, Matsuoka e Yonezawa [ICH92]

Estas implementações apresentadas serviram para uma visão do atual panorama a respeito de reflexão computacional em relação às linguagens de programação orientadas a objetos. Até o presente momento não foi encontrado nenhum trabalho que estendesse um suporte à reflexão computacional aos ambientes de desenvolvimento visual mais conhecidos, tais como Visual Basic e Visual C++ (da Microsoft) e Delphi, JBuilder e C++Builder (da Borland). A discussão sobre estes ambientes é assunto para o próximo capítulo.

Capítulo 3
AMBIENTES DE DESENVOLVIMENTO VISUAL

Em 1995, Bill Gates, presidente da Microsoft, comentou que a orientação a objetos será a mais importante tecnologia de software emergente dos anos 90. De fato, a julgar pela disponibilidade atual de aplicações que utilizam a orientação a objetos, há de se concordar com ele. Infelizmente, as técnicas orientadas a objetos, sozinhas, não são suficientes para sustentar o mundo do desenvolvimento do software. Elas devem ser combinadas com outras tecnologias, tais como, ferramentas CASE, geradores de código, desenvolvimento baseado em repositórios, programação visual, bancos de dados orientados a objetos, linguagem não procedimental e tecnologia cliente-servidor, entre outras [MAR95].

Podem ser acrescentados a esta lista os ambientes de desenvolvimento visual como mais uma ferramenta de suporte à confecção de software. Observa-se, porém, que a evolução das tecnologias citadas por Martin não ocorreu conforme o previsto, comprovado pelo fato destes ambientes não terem sidos citados.

Historicamente falando, o surgimento destes ambientes como uma nova metodologia de engenharia de software deu-se em fins dos anos 70 e início dos 80, quando pesquisadores do Centro de Pesquisas da Xerox, em Palo Alto, Califórnia, desenvolveram a linguagem de programação Smalltalk, que utilizava pela primeira vez o conceito de classes e objetos, além de possuir uma interface gráfica baseada em janelas e ícones que se sobrepunham de modo a organizar as saídas de múltiplos programas. [ALV97, JON94, SWA92]. Algumas companhias seguiram os passos da Xerox, como a Apple Corporation, que rapidamente perceberam as vantagens práticas das interfaces gráficas para usuários (GUI – *Graphical User Interface*), lançando os

computadores Lisa e Macintosh. Nesta época, os usuários dos PCs não tinham ainda uma GUI para os sistemas baseados no MS-DOS. Somente após a versão 3 do Windows em meados de 1989 é que começou a haver uma competição entre os usuários de PCs e Macs em busca das maravilhas dos ambientes gráficos [SWA92].

As regras ditadas pelo mercado fizeram com que surgissem vários ambientes de desenvolvimento como uma evolução natural das linguagens até então voltadas ao ambiente MS-DOS. Estes ambientes, de certa forma, afetaram a forma de se pensar em termos de programação. Pelas suas características, pode-se ter a falsa idéia de que uma aplicação inteira seja criada para o usuário. Na realidade, estas ferramentas facilitam a árdua tarefa de montagem da interface gráfica da aplicação através da incorporação de componentes visuais e não visuais disponíveis no ambiente. E este era o principal obstáculo para que sistemas textuais migrassem para o mundo dos ícones e janelas.

3.1. Alguns ambientes

Dentre os vários ambientes de desenvolvimento visual orientados a objetos existentes, destacam-se, agora, alguns. O critério utilizado para esta escolha reside no fato de serem utilizados tanto em ambientes acadêmicos quanto em corporativos.

3.1.1. VisualWorks

Desenvolvido pela ParcPlace System em 1994, como um ambiente de desenvolvimento para o Smalltalk, com ferramentas tanto para a montagem das GUIs quanto para o código Smalltalk. Um *canvas* permite a construção de interfaces gráficas através da seleção de elementos de uma palheta. O Smalltalk possui um estilo arquitetônico de software denominado MVC (*Model-View-Controller*), na qual uma aplicação é dividida em três partes: *Model* que diz respeito a estrutura de armazenamento e tratamento dos dados da aplicação; *View* que corresponde a elementos visuais da aplicação; e *Controller*, responsável pela entrada de dados. *Model*, *View* e *Controller* são classes que devem ser especializadas por herança. [SIL00, PAR94, PAI96].

3.1.2. Visual Basic

Ao contrário dos demais ambientes, o Visual Basic (VB), da Microsoft, é considerado uma linguagem. Foi originária do QuickBasic, um dialeto do Basic com facilidades na manipulação de código estruturado, semelhante ao C, Pascal e outras. O VB pode ser considerada uma linguagem verdadeiramente revolucionária, pois foi a primeira ferramenta de programação lançada no mercado que combinava a POO com a programação motivada por eventos, além de introduzir mudanças conceituais no desenvolvimento de aplicações do estilo "dados" para a gráfica [ENT93].

3.1.3. Delphi

Lançado pela Borland International Inc. em 1994 para competir com Visual Basic, o Delphi rapidamente tornou-se uma ferramenta bastante popular para o desenvolvimento de aplicações no Windows. Ele é baseado no Object Pascal, uma linguagem híbrida que combina o Pascal procedimental com extensões da POO. Utiliza a VCL (*Visual Component Library*) para o desenho das interfaces. Sua versão mais recente suporta também componentes CORBA [CAN96].

3.1.4. Visual C++

Desenvolvido também pela Microsoft. Engloba num mesmo produto dois sistemas completos de desenvolvimento: uma aplicação utiliza ou a API do Windows SDK (*Software Development Kit*) ou as classes do C++ da *Class Library Reference* por intermédio de ferramentas específicas como o AppWizard (gerador de código que cria o esqueleto da aplicação) e o ClassWizard (mantêm consistentes os recursos desenhados pelo AppStudio com seu código fonte) [KRU94].

3.1.5. JBuilder

Integrante da família Borland, foi lançado em 1997 para o desenvolvimento de aplicações Java independente de plataforma (Windows, Linux ou Solaris). Sua versão mais recente suporta componentes JavaBeans e CORBA.

3.1.6. C++Builder

Lançado em 1996, também pela Borland, como um suporte a aplicações escritas em C++. Sua integração com o Delphi é bastante grande. Utiliza também a VCL (*Visual Component Library*).

3.2. Características

A proposta dos ambientes de desenvolvimento visual é utilizar tanto o texto quanto a imagem para a confecção de aplicações. Desta forma, pode-se considerar que estes ambientes estão num estágio intermediário em relação às ferramentas textuais de programação (texto) e as linguagens visuais[4] (imagem). E é fácil perceber o porquê da adoção desta abordagem híbrida de texto e imagem. Uma vez que as linguagens visuais são recentes, há poucas ferramentas de suporte ao seu uso (editores, depuradores, compiladores etc.). Além do mais, as linguagens visuais ainda estão limitadas a domínios específicos no desenvolvimento de aplicações, enquanto que os ambientes de desenvolvimento visual são suportados por uma linguagem de programação tradicional que possibilita a confecção de software em qualquer domínio.

Talvez a razão da aceitação dos ambientes visuais resida no fato de proporcionar a reutilização de aplicações escritas em ferramentas textuais de programação. É claro que o grau de reutilização depende da linguagem e do estilo adotado ao se escrever o

[4] Kleyn, Michail e Hils [KLE95, MIC97, HIL92] afirmam que as linguagens visuais promovem um alto grau de abstração sobre um domínio específico de problema, na qual um programa é manipulado apenas por diagramas ao invés de texto.

código desta aplicação. As aplicações onde as classes são organizadas com uma clara separação entre funcionalidades e aspectos de interface terão um aproveitamento maior do código escrito, pois a adaptação desta aplicação no novo ambiente será menos traumática.

As figuras seguintes mostram alguns detalhes dos ambientes do Delphi, do Visual Basic e do VisualWorks[5]. Para uma rápida comparação de suas características, foram escolhidos três ambientes de fabricantes distintos. Uma comparação entre as recentes versões do Delphi e o C++Builder, por exemplo, torna-se sem sentido, pois as únicas diferenças entre os dois ambientes residem, possivelmente, no ícone do produto e na linguagem de programação associada (o Delphi usa o Object Pascal e o C++Builder o C++).

Apresentam-se, agora, algumas características comuns aos três ambientes:

3.2.1. Simplificação do uso

Simplificam de forma significativa o desenvolvimento de interfaces por permitir um rápido acesso às principais funcionalidades do ambiente, uma vez que o ambiente propicia uma personalização da apresentação destas funções ao usuário (figuras 3.1, 3.2 e 3.3).

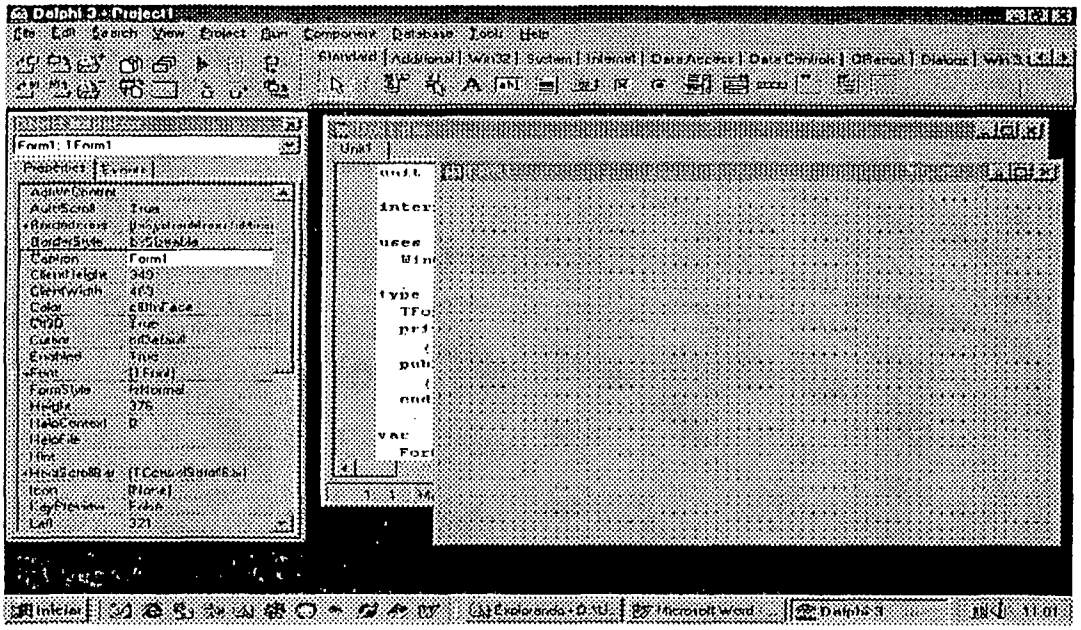
3.2.2. WYSIWYG

São as iniciais de "*What You See Is What You Get*" que, traduzindo, seria algo como "o que você vê é o que você terá". Como a interface[6] é montada pela colocação de componentes e muitas das características visuais destes componentes são notadas ainda em tempo de desenvolvimento, a forma como a interface se apresenta quando em construção é como aparecerá no momento de execução da aplicação. Isto permite um

[5] Delphi versão 5.0, Visual Basic versão 6.0 e VisualWorks versão 2.0

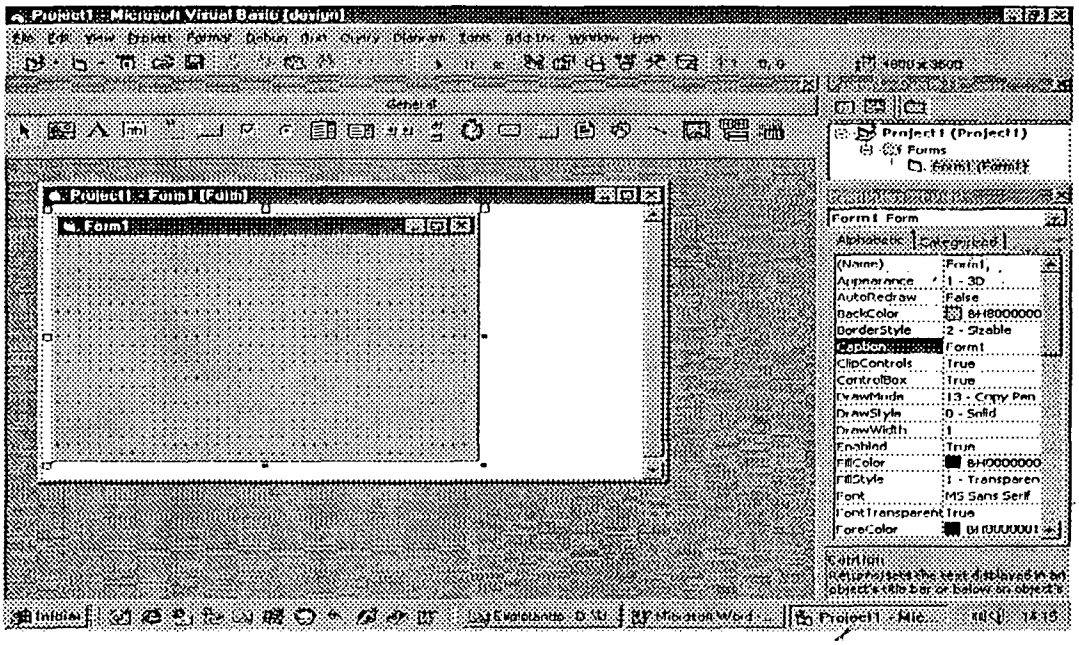
[6] No *form* para o Delphi e VB e no *canvas* para o VisualWorks

menor tempo de desenvolvimento, pois o usuário não precisa executar a aplicação para verificar como está seu desenho da interface.



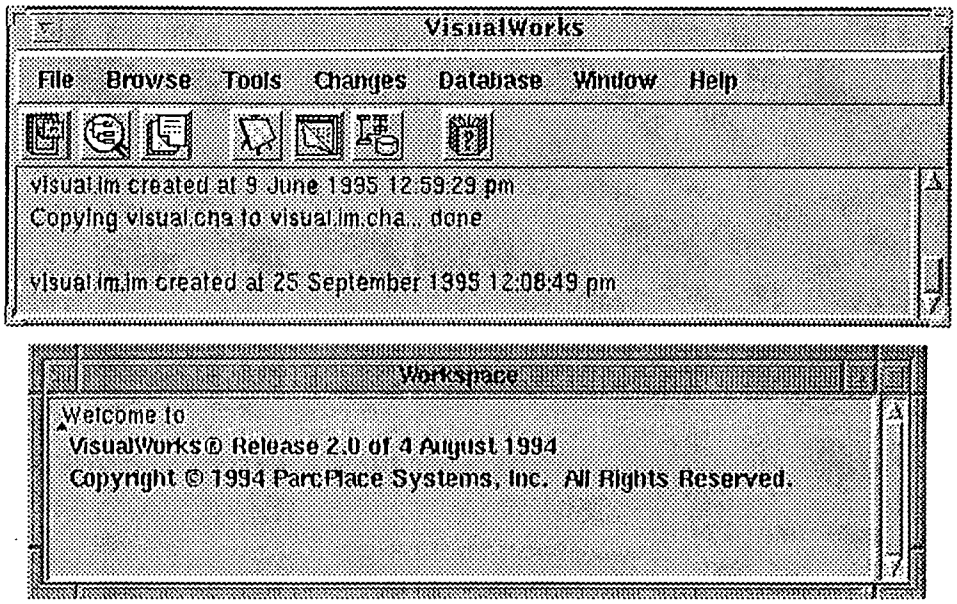
As quatro principais janelas do Delphi. A janela principal na parte superior contém os ***SpeedButtons*** (a esquerda) e a **palheta de componentes** (na direita). Abaixo, a esquerda, vê-se a do ***Object Inspector*** e ao lado desta o ***form*** (formulário) por cima da janela da ***unit***.

FIGURA 3.1: O AMBIENTE DO DELPHI



As seis principais janelas do VB. A janela principal na parte superior contém os ***SpeedButtons***. Logo abaixo está a **caixa de ferramentas** com o título de *General*. Ao lado desta está a **janela de projeto** e abaixo a **janela de propriedades**. Na parte esquerda da tela estão os controles do ***form*** (formulário).

FIGURA 3.2: O AMBIENTE DO VB

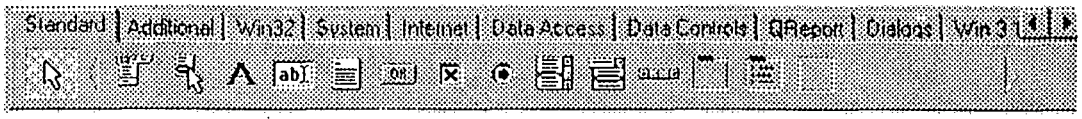


A primeira é a principal. Contém os ***SpeedButtons*** e uma área para saída textual. A segunda janela é a ***Workspace***, um local para testes de partes de código Smalltalk.

FIGURA 3.3: AS DUAS JANELAS INICIAIS DO VISUALWORKS

3.2.3. Reutilização de código e projeto

Uma vez que os elementos da interface disponibilizados pelo ambientes estão na forma de componentes, a reutilização fica garantida pelo próprio conceito de componentes. Normalmente, os ambientes de desenvolvimento visual de software permitem não apenas usar os componentes como também criar novos. Os três ambientes discutidos tratam esta questão de forma diferenciada. No Delphi e no VB é possível criar novos componentes, assim como novas classes. Já no VisualWorks somente classes podem ser criadas e reutilizadas. As figuras 3.4, 3.5 e 3.6 mostram como os ambientes apresentam os componentes.

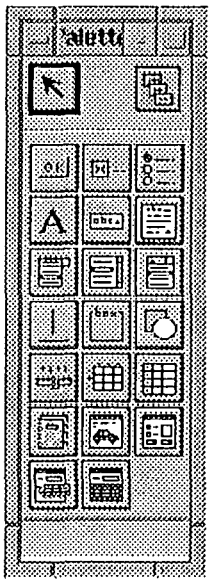


Cada página contém vários componentes, agrupados segundo suas funcionalidades ou por fornecedores. Em versões mais recentes do Delphi e do C++Builder, a Borland padronizou a estrutura dos componentes, de modo que um mesmo componente pode ser utilizado nos dois ambientes independentemente de como foram escritos (em Object Pascal ou em C++). Tem o nome de VCL (*Visual Component Library*)

FIGURA 3.4: A PALHETA DE COMPONENTES DO DELPHI

Estes são os componentes default mostrados. O usuário pode disponibilizar no ambiente outros constantes na biblioteca do VB.

FIGURA 3.5: A CAIXA DE FERRAMENTAS DO VB COM SEUS COMPONENTES



O VisualWorks não permite que sejam criados novos componentes pelo usuário. Eles podem ser apenas utilizados.

FIGURA 3.6: OS COMPONENTES DO VISUALWORKS

3.2.4. Ocultação de detalhes

Esta questão refere-se à capacidade dos ambientes em ocultarem detalhes da aplicação. Como os componentes da interface têm suas propriedades ajustadas por intermédio de janelas, o próprio ambiente encarrega-se de manter atualizados os arquivos que armazenam os valores das tais propriedades (como, por exemplo, cor, fonte dos caracteres, posição na interface etc.). De um modo geral, estes arquivos ficam ocultos durante o processo de desenvolvimento da aplicação. Com exceção do VisualWorks, que utiliza apenas um arquivo[7] de armazenamento, o Delphi[8] e o VB utilizam dois arquivos a cada interface da aplicação, em que um deles armazena a própria interface e o outro as propriedades definidas dos componente desta interface.

3.2.4. Inspeção de elementos

Os ambientes de desenvolvimento visual de aplicações suportam a inspeção não só da hierarquia de classes da linguagem e das classes criadas pelos usuário como também das propriedades atribuídas a cada componente integrante da aplicação (figuras 3.7, 3.8 e 3.9).

3.2.5. Imagem + texto

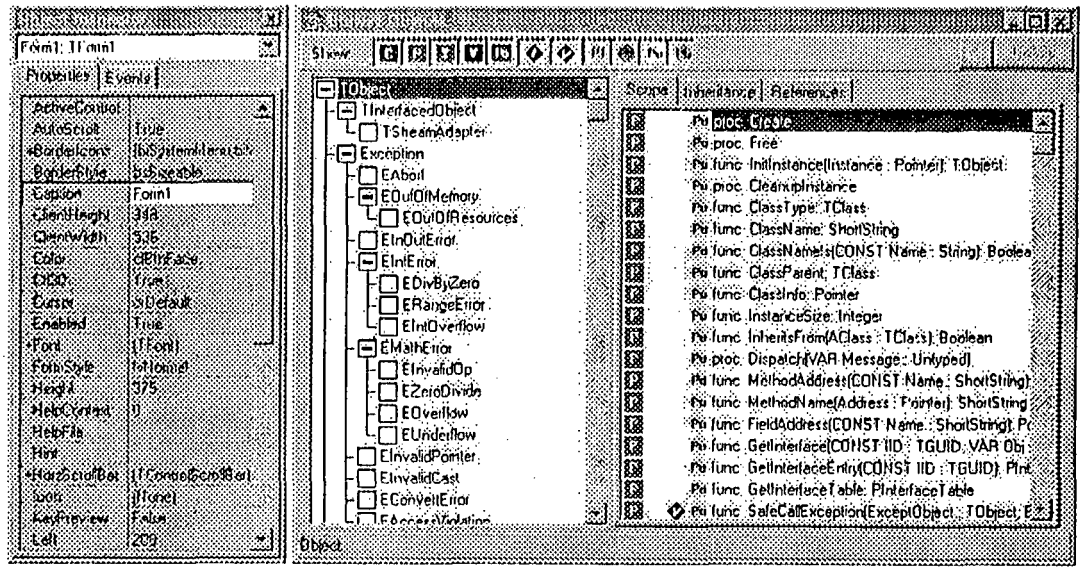
Conforme já comentado, os ambientes visuais são ambientes híbridos, ou seja, combinam aspectos da programação OOP tradicional com o uso de imagens para o desenho de interfaces. Estando os componentes acomodados nas janelas da aplicação (parte imagem), o projetista define, então, os algoritmos associados a alguns destes componentes (parte texto). Estes algoritmos possuem os mesmos elementos da programação tradicional, tais como atribuições, laços de repetição, estruturas de decisão e outras. As figuras 3.10, 3.11 e 3.12 mostram como os ambientes analisados tratam esta questão.

---

[7] Este arquivo, chamado de "imagem", contém não apenas a aplicação do usuário, mas também a biblioteca de classes e todas as definições do ambiente, tais como disposição e configuração das janelas e novos objetos Smalltalk criados.

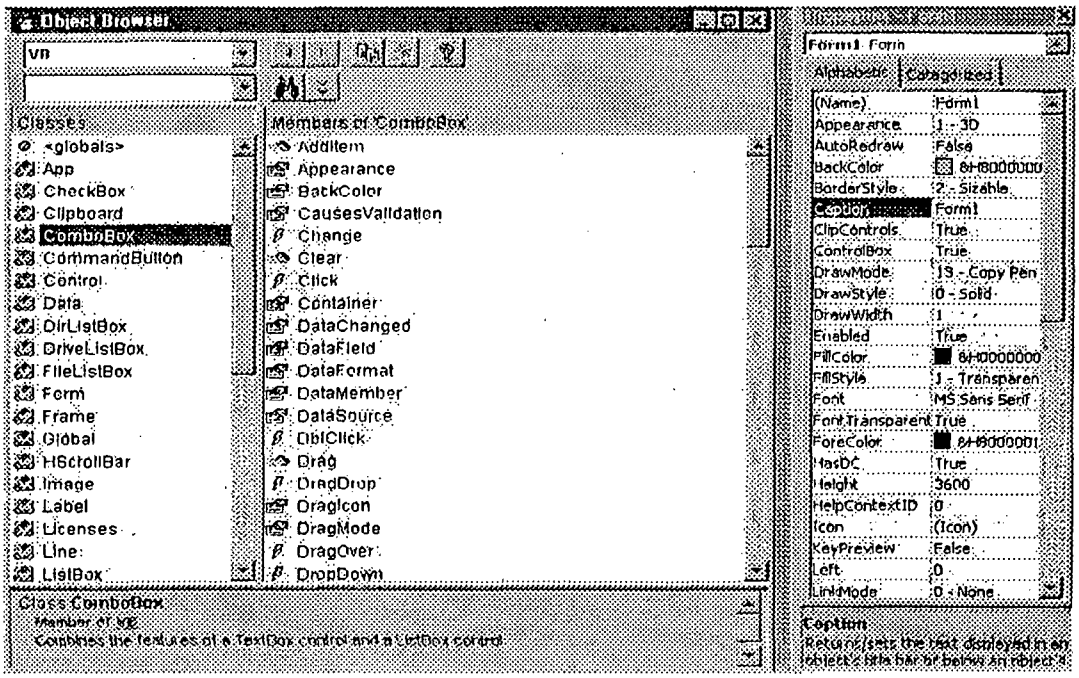
[8] As figuras A.2 e A.3 são bons exemplos de ocultação de detalhes.

O ponto forte dos ambientes de desenvolvimento visual é, sem dúvida, o suporte oferecido ao usuário no momento em que ele está escrevendo a aplicação, embora ofereçam recursos visuais de acompanhamento da execução. Nos ambientes estudados, percebe-se que o desenvolvimento visual funciona quase que da mesma maneira. É claro que há algumas diferenças no que diz respeito à criação de objetos visuais e no modo como são implementados. No entanto, mesmo com todas as suas diferenças, os ambientes visuais possuem muita coisa em comum.



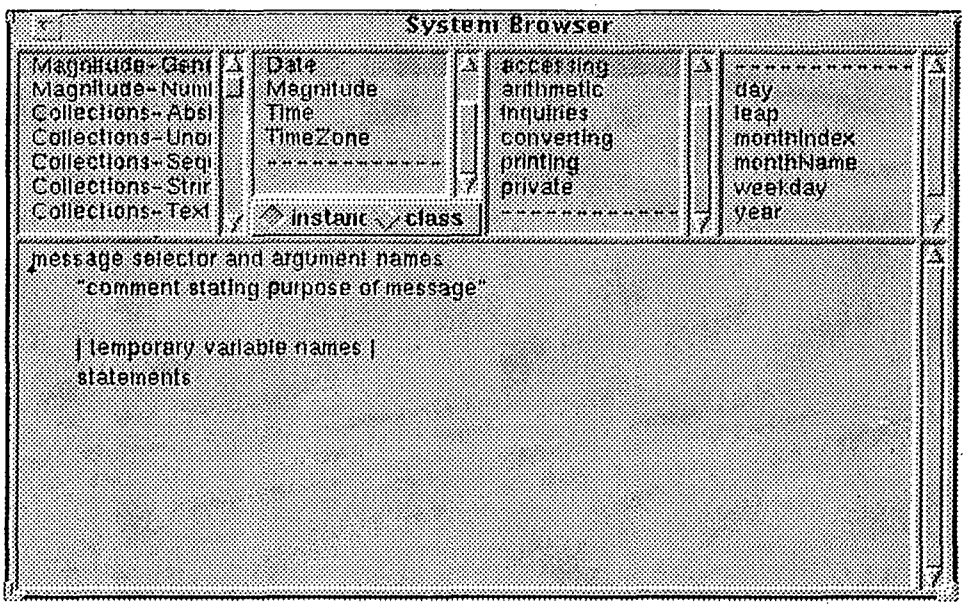
A janela ***Object Inspector*** (esquerda) permite inspecionar as propriedades e os eventos definidos para cada elemento da interface enquanto que na ***Browse Objects*** é possível verificar toda a hierarquia das classes do Object Pascal e aquelas criadas pelos usuário.

FIGURA 3.7: OS INSPETORES DO DELPHI



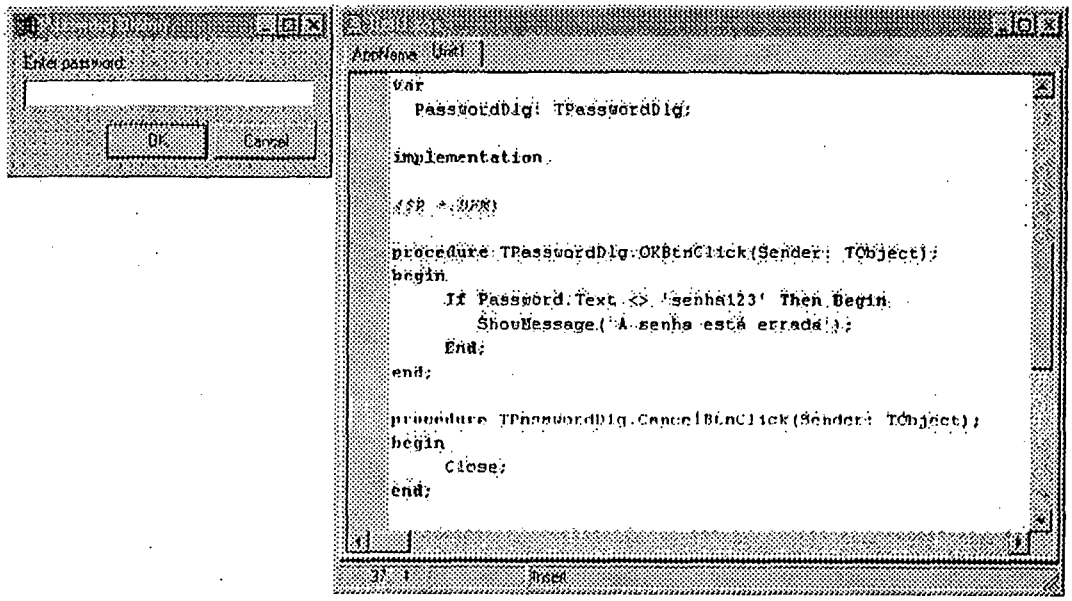
A janela ***Object Browser*** (esquerda) mostra a hierarquia de classes do VB assim como as classes do usuário e a ***Properties*** inspecionar as propriedades e os eventos definidos para cada elemento da interface.

FIGURA 3.8: OS INSPETORES DO VB



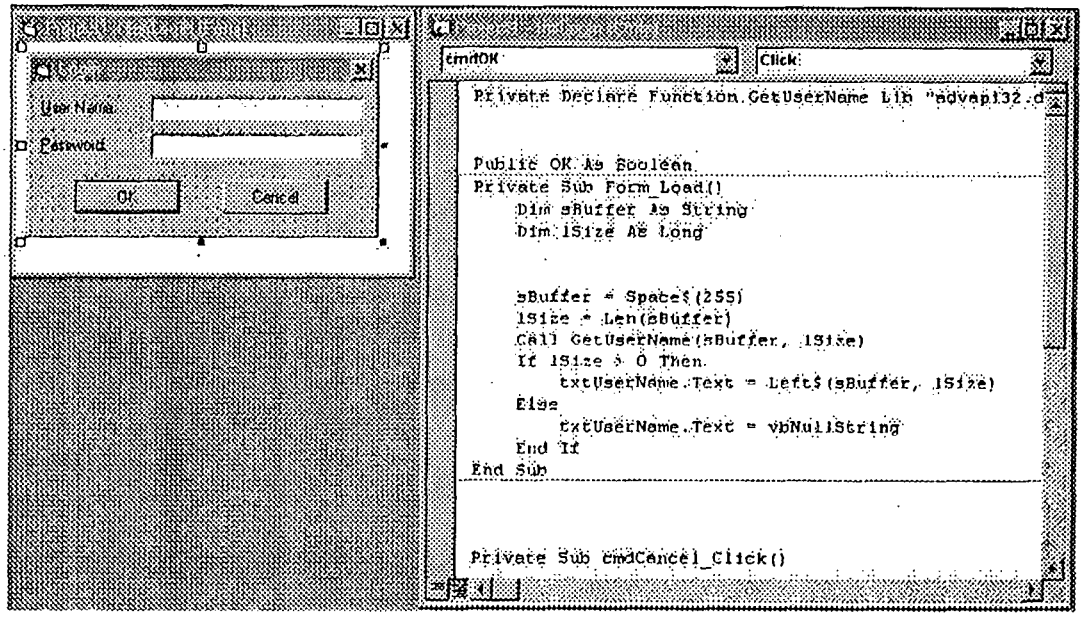
Uma janela com 5 visões. Na parte superior, a primeira coluna (*Class Categories*) apresenta as classes do sistema. A segunda (*Classes*) mostra todas as classes da categoria selecionada. Na terceira (*Message Categories*) as mensagens que a classe pode responder. Na quarta janela, os métodos. Finalmente, na parte inferior, a janela com código Smalltalk.

FIGURA 3.9: OS INSPETORES DO VISUALWORKS



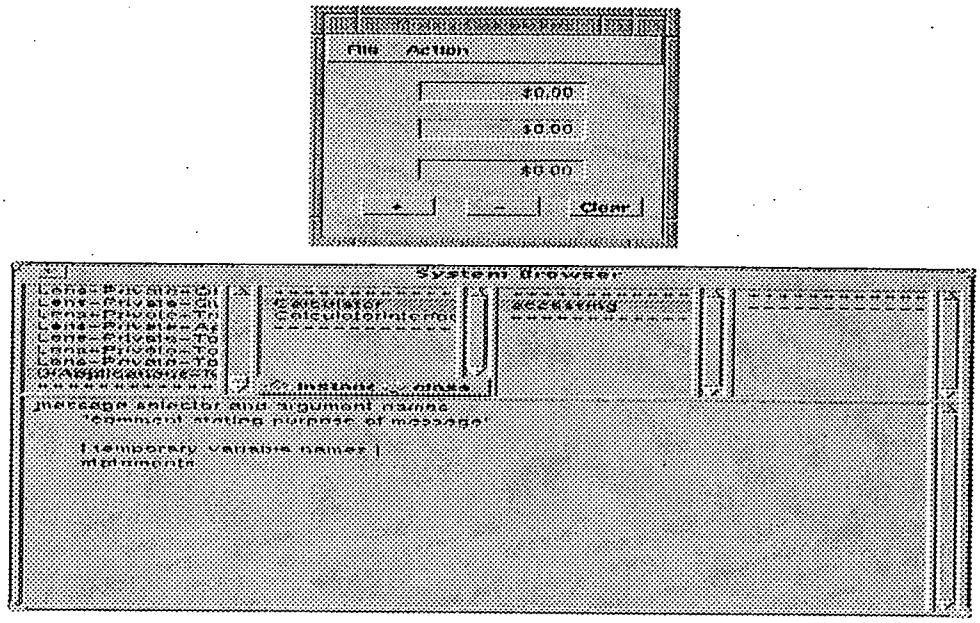
Na esquerda vê-se o *form* desenhado pelo usuário e a direita a ***unit*** com os algoritmos que serão executados ao serem pressionados os botões "OK" e "Cancel".

FIGURA 3.10: IMAGEM E TEXTO DO DELPHI



A interface desenhada pelo usuário está na janela da esquerda e o código QuickBasic está na janela da direita.

FIGURA 3.11: IMAGEM E TEXTO NO VB



Da arquitetura MVC, a classe "Calculator", do exemplo, é parte integrante do *Model* e a classe "CalculatorInterface" da *View*.

FIGURA 3.12: IMAGEM E TEXTO NO VISUALWORKS

Capítulo 4
PROPOSTA DE SUPORTE

A exploração maciça do paradigma da orientação a objetos provoca o surgimento de novas idéias no campo do desenvolvimento de software, não só com o intuito de estender funcionalidades, mas também para cobrir falhas e limitações nas abordagens existentes. Pensando em reutilização, a Engenharia de Software apresenta os *frameworks* e os componentes[9] como uma alternativa para que artefatos de software sejam reutilizáveis não apenas em termos de código como também de projeto.

Conceitualmente, um componente é uma unidade de composição com interfaces contratualmente especificadas e dependências de contexto explícitas [SZY96]. Uma interface especificada não significa ter que demonstrar como ela foi implementada, mas sim descrever um grupo de especificações de serviços do componente e sua independência é garantida pelo encapsulamento de detalhes de implementação.

Em relação aos ambientes de desenvolvimento visual de software, a forma de apresentação de suas funcionalidades é realizada por intermédio de elementos gráficos implementados principalmente por componentes. O capítulo anterior apresentou alguns ambientes e, em todos eles, observa-se a presença de componentes como uma forma de reutilização de software. As ferramentas trazem vários componentes implementados, cada um com sua especificação e funcionalidade. O papel do usuário, quando apenas os utilizam, resume-se a incorporá-los à aplicação e customizá-los conforme a necessidade, definindo propriedades e interações. Em termos de OO, este ajuste nada mais é do que

[9] *Frameworks* orientados a objetos e componentes, conforme apêndice B

uma especialização da classe que o componente representa, onde os atributos e os aspectos comportamentais desta classe são definidos por intermédio da interface do componente.

A proposta de estender recursos de reflexão computacional em ambientes de desenvolvimento visual de software deve contemplar componentes que promovam facilidades na confecção de aplicações com características reflexivas. Estas facilidades, então, devem ocorrer no momento em que a aplicação está ainda sendo desenvolvida (tempo de *design*) e não apenas quando de sua compilação ou execução (*runtime*). Portanto, a idéia é prover, em tempo de *design*, o resultado da pré-compilação através de componentes visuais.

Propõe-se, então, para um suporte básico à reflexão comportamental, a criação de quatro componentes: o Componente Reifica (CR), responsável pelo processo de reificação; o Componente Texto (CT) encarregado de prover o resultado da reificação em componentes visuais; o Componente Associa (CA), responsável em promover todo o suporte para a realização do processo de interceptação de mensagens; o Componente Execução (CE), contendo facilidades na definição de quais métodos serão reflexivos.

4.1. Componente Reifica (CR)

Para prover o processo de reificação, a proposta é criar um componente, denominado de **CR** (Componente Reifica). Antes de sua apresentação, considerar a figura 4.1. A classe `C_NB`, presente no arquivo `ArqClasseBase.xxx`, possui um único atributo `Valor` e três métodos: um construtor `Init`, dois procedimentos `Adiciona` e `Operacao` e uma função `GetValor`.

```
ArqClasseBase.xxx

Classe C_NB
          Valor : int
     Métodos
          Init
          Adiciona (i : int)
          Operacao (n : int; op : char)
          GetValor : int
Fim classe
```

FIGURA 4.1. CLASSE BASE DE EXEMPLO

Características do componente CR:

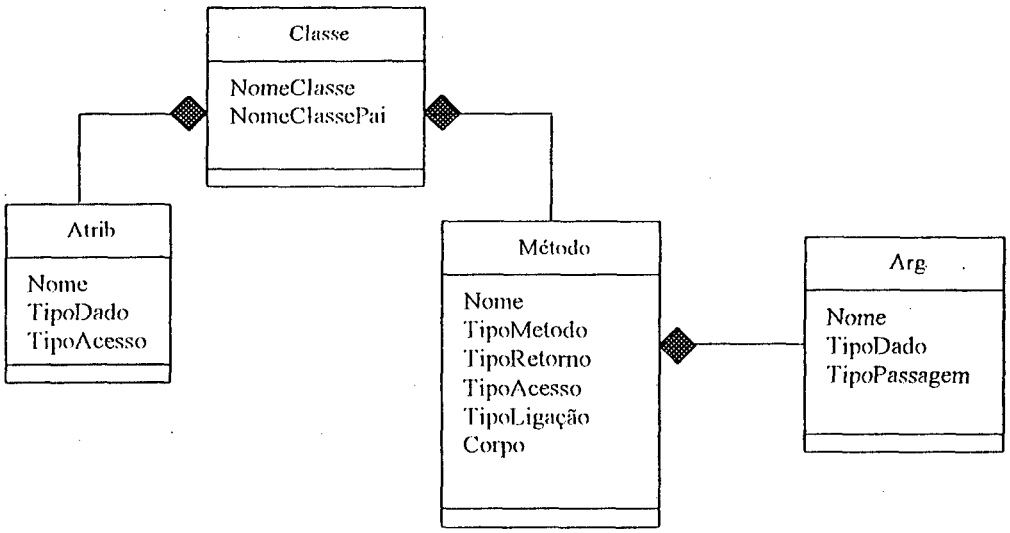
- Interface: nome da classe nível base, arquivo onde está declarada e nome do componente.

- Exemplo de interface: (conteúdos conforme figura 4.1)

Propriedade	Conteúdo
Classe base	C_NB
Arquivo da classe base	ArqClasseBase.xxx
Nome	NomeCR

- Funções: reificar as informações da classe nível base C_NB e armazená-las num conjunto de classes conforme a figura 4.2. O processo de armazenamento permanente destas classes pode utilizar os recursos de persistência da linguagem de programação do ambiente, um sistema de arquivos com uma estrutura de dados proprietária ou ainda um sistema de banco de dados. Nesta escolha deve ser considerada a questão do desempenho de leitura e gravação das informações reificadas, de forma a afetar o menos possível o desempenho da aplicação. Outra função proposta para CR é a de criar um Componente Execução (CE) para cada método da classe base. O Componente Execução é apresentado no item 4.4 deste capítulo.

- Outras características: componente não visual[10] e pode ser utilizado tanto em tempo de desenvolvimento quanto em execução.



TipoDado = básicos (inteiro, real, vetor, registros) ou definidos pelo usuário
TipoAcesso = público, privado, protegido
TipoMetodo = procedimento, função, construtor, destrutor
TipoRetorno = mesmo TipoDado
TipoLigação = dinâmico, estático
TipoPassagem = por valor, por referência, constante

FIGURA 4.2: DIAGRAMA DE CLASSES DAS META-INFORMAÇÕES

4.2. Componente Texto (CT)

A função básica deste componente é prover o resultado da reificação em componentes visuais. Suas características são:

- Interface: nome de uma instância do componente CR, nome da característica (atributos ou métodos) da classe, tipo de informação a ser mostrada de acordo com a característica escolhida, nome do componente.

[10] Um componente é dito "visual" quando possui a característica de ser visto, ainda em tempo de *design*, na forma como se apresentará em *runtime* e é dito "não visual" quando não possui esta característica (não são eles próprios visíveis em tempo de execução, mas podem gerenciar algo que é visual).

- Exemplos de interface: (conteúdos conforme figura 4.1)

(a)

Propriedade	Conteúdo
Instância CR	NomeCR
Característica	Adiciona
Tipo Informação	TipoMetodo
Nome	CompoCT1

(b)

Propriedade	Conteúdo
Instância CR	NomeCR
Característica	Adiciona
Tipo Informação	TipoAcesso
Nome	CompoCT2

(c)

Propriedade	Conteúdo
Instância CR	NomeCR
Característica	Adiciona
Tipo Informação	Nome
Nome	CompoCT3

(d)

Propriedade	Conteúdo
Instância CR	NomeCR
Característica	Adiciona
Tipo Informação	NomeClasse
Nome	CompoCT4

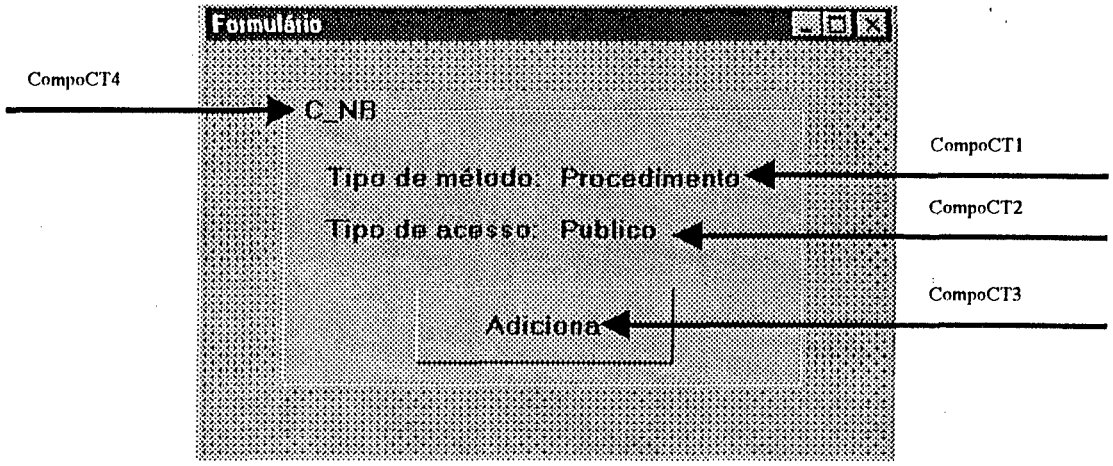
- Função: prover o aparecimento do conteúdo de tipo de informação da característica informada, obtido da instância do componente CR (que conhece toda a estrutura da classe nível base). Em sua implementação, CT será derivado de um componente visual e terá as propriedades acima acrescentadas às do componente ancestral. A figura 4.3 mostra um formulário com vários componentes CT implementados.

- Outras características: componente visual e pode ser utilizado tanto em tempo de desenvolvimento quanto em execução.

Os componentes que podem ser ancestrais de CT são aqueles padrão oferecidos pelo ambiente que possuam a capacidade de mostrar uma *string* dentro de seu próprio desenho. Um botão, por exemplo, possui um rótulo. Um painel é um componente que desenha uma moldura que pode abrigar outros componentes e este painel tem, também, um rótulo. Outros exemplos: botão de rádio, painel de grupo de botões e botões com ícone.

Os tipos de informações tratados pelo componente CT são aquelas descritas nas classes Classe, Atrib, Metodo e Arg, da figura 4.2, e dependerão da característica

escolhida em sua interface. Em outras palavras, os tipos de informações para a caraterística Atributo não são as mesmas para a característica Metodo (um atributo possui um tipo de dado e um método possui argumentos)



`CompoCT1` e `CompoCT2` são derivados de um componente "rótulo" (*label*). Já `CompoCT3`, deriva de "botão" (*button*) e `CompoCT4`, de "painel" (*panel*). "Tipo de método" e "Tipo de acesso" são componentes padrão do ambiente

FIGURA 4.3. FORMULÁRIO COM VÁRIOS COMPONENTES

4.3. Componente Associa (CA)

O terceiro componente proposto é encarregado de prover todo o suporte para a realização do processo de interceptação de mensagens, processo este indispensável para implementação do suporte da reflexão comportamental. O metanível controla a execução da classe nível base por intermédio de um meta-objeto associado, onde todas as mensagens enviadas ao objeto base serão interceptadas e desviadas ao metanível.

A figura 4.4 apresenta um exemplo de uma metaclasse `C_MN`, presente no arquivo `ArqClasseMeta.xxx`, que herda `MOP` e possui um único método chamado `ControleExecucao`. A classe `MOP` possui alguns métodos que necessitam serem sobrescritos para a especificação de como será este controle da execução (`ControleExecucao` é um deles).

```
ArqClasseMeta.xxx

Classe C_MN (MOP)
      Métodos
            ControleExecucao (metodo, args, ...) : logico override
Fim classe
```

FIGURA 4.4. METACLASSE DE EXEMPLO

Características do componente CA:

- Interface: nome da classe nível base e o arquivo onde está sua declaração, o nome da classe metanível e o arquivo onde está sua declaração, nome de uma instância do componente CR, nome do componente.

- Exemplo de interface: (conteúdos conforme figuras 4.1 e 4.3 e exemplo de interface do componente CR)

Propriedade	Conteúdo
Classe base	C_NB
Arquivo da classe base	ArqClasseBase.xxx
Classe metanível	C_MN
Arquivo da classe metanível	ArqClasseMeta.xxx
Instância Componente CR	NomeCR
Nome	NomeCA

- Funções: promover a reificação da classe base e prover o suporte para a interceptação de mensagens. Se for informada uma instância do componente CR, as informações referentes a classe base já serão conhecidas. Caso contrário, faz-se necessário informar, também, além dos dados da classe do metanível, o nome da classe base e o arquivo que contém sua declaração. Uma das tarefas do Componente Associa é a mesma do Componente Reifica, uma vez que não há a obrigatoriedade da existência na aplicação de uma instância de CR. Como a classe base precisa estar reificada no momento da realização do provimento do suporte à interceptação de mensagens, CA também realiza a reificação. Outra função proposta para CA é a de criar um Componente Execução (CE) para cada

método da classe base. O Componente Execução é apresentado no item 4.4 deste capítulo.

- Outras características: componente não visual e pode ser utilizado tanto em tempo de desenvolvimento quanto em execução.

A implementação da tarefa de prover um suporte para interceptação passa pela especialização de algumas classes da aplicação do usuário e pela criação de outras. A complexidade desta implementação depende, e muito, da linguagem de programação do ambiente, pois o fluxo de execução alterna-se entre objetos e meta-objetos (figura 2.2).

Os desafios envolvem não apenas a questão da transparência, mas também da realização de um mínimo de alterações no código escrito pelo usuário, já que, conforme discutido no capítulo anterior, os ambientes visuais possuem a característica de ocultamento de detalhes. Como o protocolo de meta-objetos necessita realizar algumas alterações internas, então que sejam feitas naquelas partes da aplicação atualizadas automaticamente pelo próprio ambiente.

De um modo geral, uma aplicação típica de um ambiente visual escrita com recursos reflexivos no controle de sua execução, passa por alguns estágios que abrangem desde a iniciativa da incorporação destes recursos até o provimento completo do controle da execução.

Num primeiro estágio, uma aplicação está, por exemplo, conforme figura 4.5. `Aplicação`[11] é responsável pelo início da execução da aplicação. Seguindo o exemplo da figura 4.1, a classe `C_NB` contém as implementações das funcionalidades da aplicação e `Interface` é a classe com o desenho da interface e especializa `C_NB`.

[11] No Delphi e no VB, por exemplo, este arquivo é denominado de "projeto" e não é uma classe. Já no VisualWorks, são atribuídos recursos a uma classe principal.

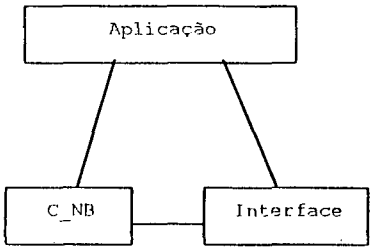


FIGURA 4.5. UMA APLICAÇÃO TÍPICA

Num segundo estágio, uma classe C_MN é criada com o intenção de controlar a execução de C_NB (figura 4.6). Conforme o exemplo da interface do Componente Associa, C_MN é a metaclasse e C_NB é a classe base. Os códigos de C_NB e C_MN podem ser observados, respectivamente, nas figuras 4.1 e 4.4.

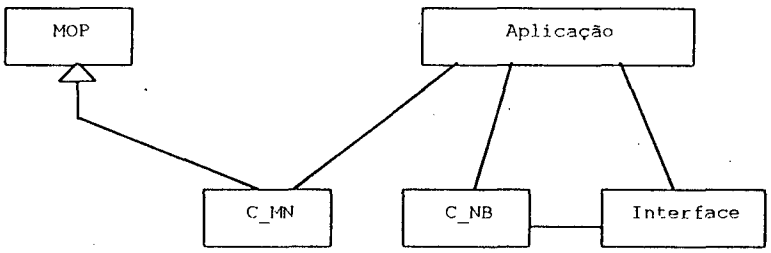
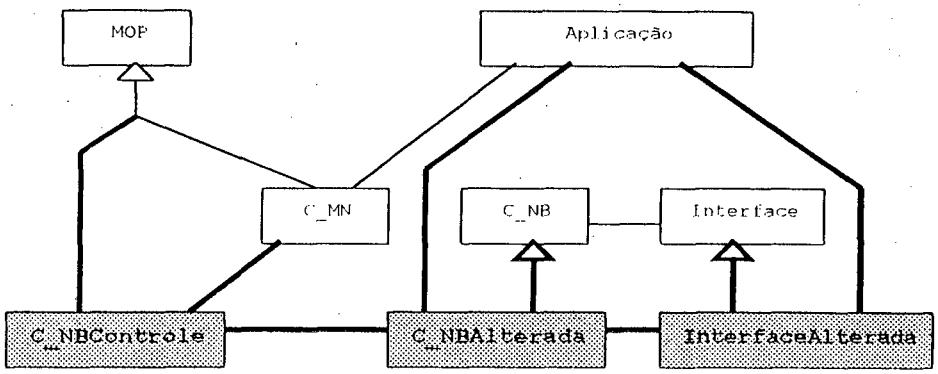
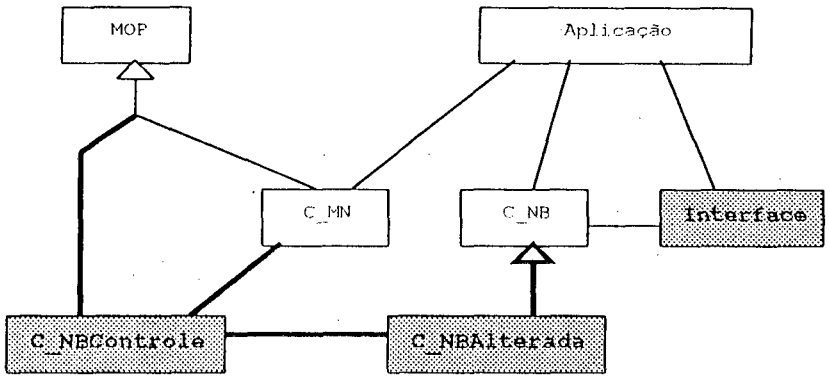


FIGURA 4.6. UMA APLICAÇÃO TÍPICA COM UMA METACLASSE

No momento em que o Componente Associa executa suas funções, o panorama da aplicação passará a um terceiro estágio. As transformações ocorridas após a associação de C_MN com C_NB podem ser de duas formas. As figuras 4.7a e 4.7b as apresentam, destacando as devidas alterações.



(A)



(B)

FIGURA 4.7. UMA APLICAÇÃO TÍPICA APÓS ASSOCIAÇÃO

As novas classes criadas pelo componente, `C_NBControle` e `C_NBAlterada`, em ambas propostas, tem as mesmas funções. `C_NBControle` controla o fluxo de execução dos métodos reflexivos entre a classe original, a metaclasse e `C_NBAlterada`, e, esta última, derivada de `C_NB`, terá seus métodos reflexivos apontados para `ControleExecucao`, implementado em `C_MN`. Desta forma, o controle de `C_NB` será de acordo com a implementação do método `ControleExecucao`, herdado de `MOP`.

As diferenças entre as propostas (A) e (B) estão, fundamentalmente, no tratamento dado a classe `Interface`, que instancia `C_NB`. Enquanto (A) propõe a criação de `InterfaceAlterada`, (B) realiza as alterações na própria `Interface`. As figuras 4.8a e 4.8b apresentam exemplos para melhor ilustrar a situação.

Ambas abordagens apresentam vantagens e desvantagens. A primeira (fig. 4.8a) tem como ponto forte permitir que Componente Associa realize adaptações na classe ascendente sem alterá-la diretamente. Isto, de certa forma, é uma postura elegante e promove um bom grau de transparência na criação do suporte à interceptação de mensagens. As mudanças em `Aplicação` (fig. 4.7a), se o ambiente permitir, são necessárias e não interferem no processo. No entanto, uma outra questão é a alta interatividade promovida pelos ambientes de desenvolvimento visual no desenho da interface, causando como conseqüência o surgimento de um ponto fraco bastante relevante: se um novo evento é acrescentado a algum componente do formulário na classe `Interface` (por exemplo, num botão que execute `ClasseNB.Adiciona(5))`, depois que o processo de criação do suporte à interceptação de mensagens de Componente Associa já tenha sido executado, será necessária uma nova invocação desta operação para que `InterfaceAlterada` contenha as atualizações. Esta desvantagem não ocorreria se todo o processo fosse executado em tempo de compilação (o que não é o caso) ou, então, se a Interface não fosse mais alterada após a atuação de CA (a criação do suporte teria de ser a última tarefa executada antes da execução da aplicação). Por outro lado, a abordagem da figura 4.8b não apresenta a desvantagem do caso (A), pois apenas troca o tipo da classe base definida. O fato de (B) alterar justamente uma classe definida pelo usuário denota uma desvantagem que não causa maiores transtornos. Por ser a única adaptação, não obrigar uma nova ativação de CA para que seja recriado o suporte à interceptação de mensagens é uma boa solução. As alterações no formulário, bastante normais em tempo de *design*, podem ser feitas livremente sem prejuízos ao suporte já definido.

```
<1>  Classe C_NB
<2>              Valor : int
<3>        Metodos
<4>              Init
<5>              Adiciona (i : int)
<6>              Operacao (n : int; op : char)
<7>              GetValor : int
<8>  Fim classe
<9>
<10> C_NB.Adiciona (i : int) { ... }
<11> C_NB.GetValor : int { ... }
```

```
<1>  Classe Interface
<2>              ClasseNB : C_NB
<3>        Métodos ...
<4>  Fim Classe
<5>
<6>  Interface.Create {
<7>  ...
<8>  ClasseNB ← C_NB.Create
<9>  ClasseNB.Adiciona (3)
<10> Show (ClasseNB.GetValor)
<11> ...}
```

```
<1>  Classe InterfaceAlterada (Interface)
<2>              ClasseNB : C_NBAlterada
<3>        Metodos ...
<4>  Fim Classe
<5>
<6>  InterfaceAlterada.Create {
<7>  ...
<8>  ClasseNB ← C_NBAlterada.Create
<9>  ClasseNB.Adiciona (3)
<10> Show (ClasseNB.GetValor)
<11> ...}
```

(A)

O Componente Associa faz uma cópia de todo o arquivo fonte que contém `Interface` para, somente após, realizar as alterações em `InterfaceAlterada` do novo arquivo (linhas <2> e <8>). Isto garante que, no arquivo cópia, todos métodos invocados de `ClasseNB` sejam de `C_NBAlterada`.

```
<1>  Classe C_NB
<2>          Valor : int
<3>      Métodos
<4>          Init
<5>          Adiciona (i : int)
<6>          Operacao (n : int; op : char)
<7>          GetValor : int
<8>  Fim classe
<9>
<10> C_NB.Adiciona (i : int) { ... }
<11> C_NB.GetValor : int { ... }
```

```
<1>  Classe Interface
<2>          ClasseNB : C_NBAlterada      // era C_NB
<3>      Métodos ...
<4>  Fim Classe
<5>
<6>  Interface.Create {
<7>  ...
<8>  ClasseNB ← C_NBAlterada.Create    // era C_NB.Create
<9>  ClasseNB.Adiciona (3)
<10> Show (ClasseNB.GetValor)
<11> ...}
```

(B)

As únicas alterações provocadas pelo Componente Associa são nas linhas <2> e <8> de `Interface`.

FIGURA 4.8: OPÇÕES DE TRATAMENTO DA CLASSE INTERFACE

Das alternativas apresentadas no tratamento da classe que instancia a classe base, a (B) parece ser a mais apropriada, no contexto destes ambientes de programação. Como muitas das tarefas são realizadas em tempo de *design*, é importante haver a preocupação de minimizar o esforço do usuário na execução de atividades que não envolvam reflexos visuais imediatos, como é o caso de Componente Associa.

Abaixo, um exemplo completo é mostrado e discutido no sentido de ilustrar o suporte proposto por CA baseado na alternativa (B). Na figura 4.9, a aplicação encontra-se em seu primeiro estágio, conforme recentemente descrito e, em 4.10, já está com a estrutura da interceptação concluída.

```
<1>  Classe Interface
<2>              ClasseNB : C_NB
<3>  Fim Classe
<4>
<5>  Interface.Create {
<6>  ...
<7>  ClasseNB ← C_NB.Create
<8>  ClasseNB.Adiciona (3)
<9>  Show (ClasseNB.GetValor)
<10> ...}
```

```
<1>  Classe C_NB
<2>              Valor : int
<3>        Métodos
<4>              Init
<5>              Adiciona (i : int)
<6>              Operacao (n : int; op : char)
<7>              GetValor : int
<8>  Fim classe
<9>
<10> C_NB.Adiciona (i : int) { ... }
<11> C_NB.GetValor : int { ... }
```

FIGURA 4.9. UMA APLICAÇÃO SEM INTERCEPTAÇÃO DE MENSAGENS

A aplicação da figura 4.9 não possui nenhum mecanismo de interceptação de mensagens. Então, quando a interface invoca um método de `ClasseNB`, está, de fato, invocando-o de `C_NB`, pois a classe base não tem nenhuma metaclasse associada para o controle de sua execução.

A figura 4.10 mostra as novas classes criadas pelo algoritmo do Componente Associa. As linhas <2> e <7> de `Interface` mostram que a classe instanciada é, agora, `C_NBAlterada`, e não mais `C_NB`, como em 4.9. Já a classe `C_NBAlterada` tem todos os métodos herdados de `C_NB` apontados para o método `ControleExecucao`, de uma instância de `C_MN` (<10> e <15>). Desta forma, o usuário terá a impressão de estar invocando `Adiciona` e `GetValor` de `C_NB`, enquanto que, na verdade, será de `C_NBAlterada`. Na metaclasse `C_MN` está a implementação de `ControleExecucao`, sobrescrito de MOP (linha <3>).

```
<1>  Classe Interface
<2>          ClasseNB : C_NBAlterada    // era C_NB
<3>  Fim Classe
<4>
<5>  Interface.Create {
<6>  ...
<7>  ClasseNB ← C_NBAlterada.Create     // era C_NB.Create
<8>  ClasseNB.Adiciona (3)
<9>  Show (ClasseNB.GetValor)
<10> ...}
```

```
<1>  Classe C_NBAlterada (C_NB)
<2>       Métodos
<3>            ...
<4>            Adiciona (i : int)
<5>            GetValor : int
<6>  Fim classe
<7>
<8>  C_NBAlterada.Adiciona (i : int) {
<9>  ...
<10>    objMN.ControleExecucao(Adiciona, i)
<11> ...}
<12>
<13> C_NBAlterada.GetValor : int {
<14> ...
<15>    objMN.ControleExecucao(GetValor)
<16> ...}
```

```
<1>  Classe C_MN (MOP)
<2>       Métodos
<3>            ControleExecucao (metodo, args) : lógico override
<4>  Fim classe
<5>
<6>  C_MN.ControleExecucao (metodo, args) : logico {
<7>  ...
<8>  se metodo = "GetValor" entao
<9>       objControle.Executa(metodo, args)
<10> fim se
<11> ...}
```

```
<1>  Classe C_NBControle (MOP)
<2>       Metodos
<3>            Executa (metodo, args) : lógico override
<4>  Fim classe
<5>
<6>  C_NBControle.Executa (metodo, args) : logico {
<7>  ...
<8>  se metodo = "GetValor" entao
<9>       C_NB.Adiciona(metodo, args)
<10> fim se
<11> ...}
```

FIGURA 4.10. UMA APLICAÇÃO COM INTERCEPTAÇÃO DE MENSAGENS

As linhas <8>, <9> e <10> mostram o tipo de controle aplicado aos métodos reflexivos de C_NB. Em C_NBControle, Executa é sobrescrito de MOP (<3>) e, em sua implementação, está a invocação do método original da classe.

Ainda da figura 4.10, o fluxo de execução de uma mensagem enviada a um objeto é, por exemplo:

1. ClasseNB.GetValor (linha <9>)
2. Linha <13> de C_NBAlterada
3. Em C_MN, linha <6>.
4. Linha <8> de C_NBControle.

4.4. Componente Execução (CE)

O quarto componente proposto possui a função de permitir a definição de quais métodos serão reflexivos. A idéia é criar um componente CE para cada método da classe base já reificada, onde a responsabilidade desta tarefa de criação será de Componente Reifica ou de Componente Associa.

Características do componente CE:

- Interface: *status* executa, nome do componente.
- Exemplo de interface: (conteúdos conforme figura 4.1)

Propriedade	Conteúdo
Executa	Sim/Não
Nome	NomeCE

- Função: permite definir quais métodos da classe base serão reflexivos.
- Outras características: componente não visual e pode ser utilizado tanto em tempo de desenvolvimento quanto em execução.

Após o processo de criação de uma instância para cada método de C_NB (figura 4.1), observam-se os seguintes componentes CE criados:

```
NomeCRInit      :  TipoCompoCE
NomeCRAdiciona  :  TipoCompoCE
NomeCROperacao  :  TipoCompoCE
NomeCRGetValor  :  TipoCompoCE
```

Propõe-se, ainda, que o nome de cada instância seja composto pelo nome do componente que as criou (NomeCR ou NomeCA) , acrescido pelo nome do método.

Concluindo, este capítulo propôs a criação de quatro componentes para um suporte básico à reflexão comportamental: Componente Reifica (CR), Componente Texto (CT), Componente Associa (CA) e Componente Execução (CE). Em termos de utilização, pode-se sintetizá-los com algumas considerações:

- Para o suporte do controle da execução, apenas CA é suficiente, pois também promove a reificação da classe base, sendo, então, independente de CR.

- Os componentes CR e CA promovem a reificação. Portanto, suas instâncias são indispensáveis para CE (permitir a definição dos métodos reflexivos)

- Apenas CT é um componente visual, o restante é não visual. Todos eles podem ser utilizados em tempo de projeto e em tempo de execução.

Capítulo 5
IMPLEMENTAÇÃO

Foi apresentado, no capítulo anterior, uma proposta de suporte à reflexão computacional baseada em quatro componentes. Aqui, neste capítulo, apresenta-se uma implementação para o ambiente Delphi, denominada de ***OP*MOP**[12], de acordo com esta proposta.

Lançado pela Borland International Inc. em 1994, o Delphi rapidamente tornou-se uma ferramenta bastante popular para o desenvolvimento de aplicações no Windows. Ele é baseado no Object Pascal, uma linguagem híbrida que combina o Pascal procedimental com extensões da POO. De suas características[13], a mais relevante, no contexto deste trabalho, é a presença de componentes, nativos e de terceiros, numa biblioteca denominada VCL (*Visual Component Library*). Ela é composta de classes não só relacionadas com o desenho da interface, como também para fins gerais, além de conter classes que não são componentes. A figura 3.4 traz a palheta de componentes do Delphi e a 3.7 mostra a janela Browse Inspector com parte da hierarquia de classes da VCL. Cada página da palheta agrupa os componentes de acordo com suas funcionalidades ou, então, oriundos de um mesmo fornecedor. Desta forma, a palheta abriga os componentes padrão do Delphi e aqueles desenvolvidos por terceiros.

A implementação *OP*MOP apresenta duas partes distintas. Uma delas, a *unit* uMOP, contém funções de conversão de tipos e definições de novos tipos de dados. A outra, contém os componentes Delphi. Ambas são detalhadas a seguir.

---

[12] O nome ***OP*MOP** vem de OP (Object Pascal) e MOP (*MetaObject Protocol*)

[13] O apêndice A traz maiores detalhes do ambiente Delphi.

5.1. A *unit* uMOP

Além de conter novos tipos de dados específicos (escalares, registros, ponteiros, listas e classes) para o uso dos componentes *OP*MOP, a *unit* uMOP contém algumas funções de conversão de tipos entre escalares e *strings*. Os nomes dos identificadores procuram seguir, dentro do possível, o mesmo padrão Delphi, conforme quadro 5.1.

Exemplo de identificador	Comentário
TE_TipoDado	TE → Tipo Escalar
TL_Atributos	TL → Tipo Lista
TP_Atributos	TP → Tipo Ponteiro
TR_Atributos	TR → Tipo Registro
TMOPAtributos	TMOP → Tipo Classe
TMOP_Metodos	TMOP → argumentos dos métodos da classe TMOPExemplo

QUADRO 5.1: PADRÃO DE NOMES DOS IDENTIFICADORES

A figura 5.1, por sua vez, apresenta os tipos escalares presentes nas demais estruturas de dados e também nos componentes. Notar que TE_TipoDado e TE_TipoRetorno não são exatamente escalares. O primeiro refere-se ao tipo de dado de cada argumento do método e o segundo é o tipo de dado de retorno de um método função. Neste caso, ambos tipos estão sendo tratados como *string*.

```
TE_TipoDado     = String
TE_TipoRetorno  = String
TE_TipoLigacao  = (tlVirtual, tlOverride, tlNenhum)
TE_TipoAcesso   = (taPublic, taPrivate, taPublished, taProtected)
TE_TipoCarac    = (tcAtributo, tcMetodo)
TE_TipoMetodo   = (tmProcedure, tmFunction, tmConstructor, tmDestructor)
TE_TipoPassagem = (tpVar, tpConst, tpNenhum)
TE_TipoInfo     = (tiNomeClasse, tiNomeClassePai,tiNome, tiId,
                   tiTipoMetodo, tiTipoDado,    tiTipoAcesso, tiTipoCarac,
                   tiTipoRetorno,tiTipoLigacao, tiCabecOriginal)
```

FIGURA 5.1: TIPOS ESCALARES DO *OP*MOP

Também presente na *unit* uMOP, as funções da figura 5.2 servem para conversão de tipos entre escalares e *strings*. A presença delas justifica-se pelo uso constante de *strings* que representam valores de escalares. Internamente, as informações ficam armazenados em forma de escalares. Já nas interfaces dos componentes, são apresentadas opções em forma de lista de *strings* que representam escalares.

```
Function EscalarTipoDado     (texto : String) : TE_TipoDado;
Function EscalarTipoRetorno  (texto : String) : TE_TipoRetorno;
Function EscalarTipoLigacao  (texto : String) : TE_TipoLigacao;
Function EscalarTipoAcesso   (texto : String) : TE_TipoAcesso;
Function EscalarTipoCarac    (texto : String) : TE_TipoCarac;
Function EscalarTipoMetodo   (texto : String) : TE_TipoMetodo;
Function EscalarTipoPassagem(texto : String) : TE_TipoPassagem;
Function EscalarTipoInfo     (texto : String) : TE_TipoInfo;

Function TextoTipoDado     (escalar : TE_TipoDado)     : String;
Function TextoTipoRetorno  (escalar : TE_TipoRetorno)  : String;
Function TextoTipoLigacao  (escalar : TE_TipoLigacao)  : String;
Function TextoTipoAcesso   (escalar : TE_TipoAcesso)   : String;
Function TextoTipoCarac    (escalar : TE_TipoCarac)    : String;
Function TextoTipoMetodo   (escalar : TE_TipoMetodo)   : String;
Function TextoTipoPassagem(escalar : TE_TipoPassagem) : String;
Function TextoTipoInfo     (escalar : TE_TipoInfo)     : String;
```

FIGURA 5.2: FUNÇÕES DE CONVERSÃO ENTRE ESCALAR E *STRING*

Outra característica da implementação *OP*MOP é conter uma classe TMOPExecucao (figura 5.3), que provê o controle da execução dos métodos. Os exemplos apresentados mais adiante mostrarão como poderá ser utilizada. Por enquanto, serão apenas comentadas as funções dos métodos disponíveis:

- **InterceptaAntes**: é um método virtual que deve ser implementado na metaclasse. O argumento indica o método que está sendo executado. Depois de concluído o suporte para a interceptação de mensagens, é o primeiro método invocado a cada método da classe base.
- **InterceptaDepois**: também é um método virtual implementado na metaclasse. É o último método invocado a cada método da classe base. Em outras palavras, InterceptaAntes e InterceptaDepois "envolvem" o código original de cada método.
- **ChamaTodas**: não deve ser invocado diretamente pelo usuário. É utilizado pelo *OP*MOP para o gerenciamento do controle da execução.

- **Executa:** responsável pela execução do método indicado como argumento. Deve ser invocado da metaclasse. Sem esta invocação, o método do argumento não será executado pela classe controladora.
- **ControleExecução:** também virtual, com implementação na metaclasse. É um método sempre referenciado pela classe controladora (C_NBControle, do exemplo da figura 4.7), criada pelo suporte de interceptação de mensagens. Todos os métodos da classe base apontam para ControleExecucao. Normalmente, sua implementação contém a invocação de Executa.

```
TMOP_Args     = Variant;
TMOP_Arg      = Variant;
TMOP_Metodo   = TR_Metodos;
TMOPExecucao = Class
      Procedure InterceptaAntes  (Metodo : TMOP_Metodo); Virtual;
      Procedure InterceptaDepois (Metodo : TMOP_Metodo); Virtual;
      Function  ChamaTodas       (Metodo : TMOP_Metodo; Args : TMOP_Args):
                                                        Boolean; Virtual;
      Function  Executa          (Metodo : TMOP_Metodo; Args : TMOP_Args):
                                                        Boolean; Virtual;
      Function  ControleExecucao (Metodo : TMOP_Metodo; Args : TMOP_Args):
                                                        Boolean; Virtual;
End;
TMOP = Class (TMOPExecucao);
```

FIGURA 5.3: TIPOS PARA O CONTROLE DA EXECUÇÃO

5.2. Os componentes de *OP*MOP

A proposta de suporte à reflexão computacional, discutida no capítulo anterior, apresenta quatro componentes básicos: Componente Reifica (CR), Componente Texto (CT), Componente Associa (CA) e Componente Execução (CE). Enquanto CR, CA e CE não prevêem nenhuma classe ancestral, CT é derivado de algum componente visual padrão do ambiente. Então, no momento de sua implementação, faz-se necessária uma análise da hierarquia de classes para a escolha deste ancestral. Como o ambiente Delphi

exige que todo componente seja derivado de alguma classe, a decisão sobre onde os novos componentes *OP*MOP encaixam-se nesta estrutura de classes passa por um estudo detalhado da VCL.

No Delphi, todo componente não visual deve herdar TComponent. Já os visuais, dependem da função que exercem. Neste sentido, as decisões de implementação estão descritas no quadro 5.2.

Proposta	Implementação *OP*MOP
Componente Reifica (CR)	TMOPReifica
Componente Execução (CE)	TMOPExecuta
Componente Associa (CA)	TMOPAssocia
Componente Texto (CT)	grupo *string*: TMOPLabel, TMOPButton, TMOPCheckBox, TMOPRadioButton, TMOPGroupBox, TMOPPanel, TMOPBitBtn, TMOPSpeedButton grupo *lines*: TMOPMemo, TMOPListBox, TMOPComboBox, TMOPRadioGroup

QUADRO 5.2: COMPONENTES IMPLEMENTADOS DA PROPOSTA

Os três primeiros componentes herdam TComponent e para o único componente visual (CT), dois grupos de componentes *OP*MOP foram criados para apresentar as meta-informações da classe base: o "grupo *string*", que utiliza uma *string*, e o "grupo *lines*", utilizando uma lista de *strings*. Enquanto os componentes do grupo *string* apresentam um tipo de informação única (por exemplo, o tipo de acesso de um certo método da classe base), os do grupo *lines* mostram uma lista (por exemplo, a lista de todos os métodos).

No total, foram implementados quinze componentes, sendo que apenas TMOPExecuta não está presente na palheta (figura 5.4). Os visuais estão assim distribuídos:

- grupo *string*: TMOPLabel, TMOPButton, TMOPCheckBox, TMOPRadioButton, TMOPGroupBox, TMOPPanel, TMOPBitBtn, TMOPSpeedButton

- grupo *lines*: TMOPMemo, TMOPListBox, TMOPComboBox, TMOPRadioGroup

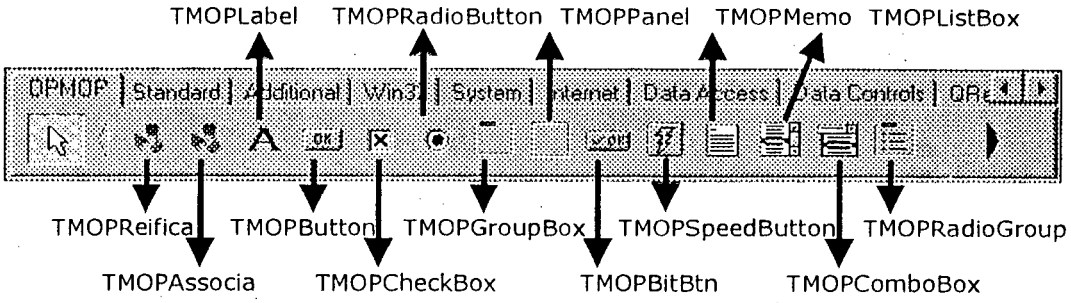


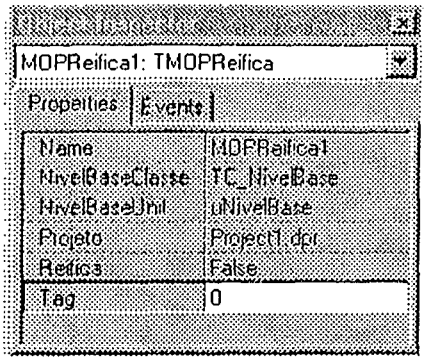
FIGURA 5.4: A PALHETA DE COMPONENTES DE *OP*MOP

As características dos novos componentes são apresentadas agora com maiores detalhes. A questão de como usá-los, no entanto, será abordada na seqüência, por intermédio de exemplos.

5.2.1 O componente TMOPReifica

Baseado no Componente Reifica (CR) da proposta, possui como função promover o processo de reificação. Uma vez informadas as propriedades Projeto, NivelBaseClasse e NivelBaseUnit, a reificação inicia-se quando Reifica for *True* (figura 5.5).

As informações da classe base ficam armazenadas no atributo público Classe. A figura 5.6 apresenta a declaração de TMOPClasse e, as seguintes, detalham as classes dos atributos, dos métodos e dos argumentos. A implementação de TMOPClasse difere um pouco do diagrama de classes proposto pela figura 4.2, pois a classe TMOPArgs, que trata dos argumentos, não está em TMOPMetodos. Esta alteração, no entanto, pouco afeta a utilização e a compreensão de TMOPClasse, uma vez que os métodos de busca implementados facilitam o acesso aos argumentos dos métodos correspondentes.




```
TMOPReifica = Class(TComponent)
  ...
    Public
      Classe : TMOPClasse;
    Published
      Property Projeto          : String;
      Property NivelBaseUnit    : String;
      Property NivelBaseClasse  : String;
      Property Reifica          : Boolean;
  ...
End;
```

A esquerda, a interface do componente apresentada na janela Object Inspector. No caso, MOPReifica1 é uma instância de TMOPReifica. As propriedades Name e Tag são herdadas de TComponent. No quadro da direita, a declaração simplificada do componente.

FIGURA 5.5: O COMPONENTE TMOPREIFICA

```
TMOPClasse = Class
               Nome       : String;
               NomePai    : String;
               Atributos  : TMOPAtributos;
               Metodos    : TMOPMetodos;
               Args       : TMOPArgs;
             End;

TMOP_Classe = Class (TMOPClasse);
```

FIGURA 5.6: A CLASSE TMOPCLASSE

```
TL_Atributos  = TList;

TP_Atributos  = ^TR_Atributos;

TR_Atributos  = Record
                  Id          : Byte;
                   Nome       : String;
                   TipoDado   : TE_TipoDado;
                   TipoAcesso : TE_TipoAcesso;
                End;

TMOPAtributos = Class
                  NroItens   : Byte;
                  Atributos  : TL_Atributos;

                  Function BuscaAtributoId (Id : Byte): TP_Atributos
                 Function BuscaAtributoNome (Nome : String): TP_Atributos
               End;
```

FIGURA 5.7: A CLASSE TMOPATRIBUTOS

```
TL_Metodos  = TList;

TP_Metodos  = ^TR_Metodos;

TR_Metodos  = Record
                Id           : Byte;
                Nome         : String;
                TipoMetodo   : TE_TipoMetodo;
                TipoRetorno  : TE_TipoRetorno;
                TipoLigacao  : TE_TipoLigacao;
                TipoAcesso   : TE_TipoAcesso;
                CabecOriginal: String;
              End;

TMOPMetodos = Class
                NroItens : Byte;
                Metodos  : TL_Metodos;

                Function BuscaMetodoId (Id : Byte)       : TP_Metodos;
                Function BuscaMetodoNome (Nome : String) : TP_Metodos;
              End;
```

FIGURA 5.8: A CLASSE TMOPMETODOS

```
TL_Args  = TList;

TP_Args  = ^TR_Args;

TR_Args  = Record
             IdMetodo     : Byte;
             Id           : Byte;
             Nome         : String;
             TipoDado     : TE_TipoDado;
             TipoPassagem : TE_TipoPassagem;
             ArgsOriginal : String;
           End;

TMOPArgs = Class
             NroItens     : Byte;
             Args         : TL_Args;

             Function BuscaArgIdMetodo (Id : Byte) : TP_Args;
           End;
```

FIGURA 5.9: A CLASSE TMOPARGS

5.2.2 O componente TMOPAssocia

Baseado no Componente Associa (CA) da proposta, TMOPAssocia é encarregado de prover todo o suporte do processo de interceptação de mensagens, implementado de acordo com a alternativa (B), da figura 4.8. Na figura 5.10, as

propriedades relacionadas à classe base podem ser obtidas de duas formas: preenchendo-as conforme TMOPReifica ou informando-se uma instância de TMOPReifica na propriedade ObjetoReifica. As propriedades MetaNivelUnit e MetaNivelClasse referem-se à metaclasse. Toda estrutura para a interceptação de mensagens é definida quando Associa for *True* e é desfeita quando retornar a *False*.

Outra tarefa realizada pelo componente é a reificação, tal qual TMOPReifica, necessária para a tarefa da associação. Este recurso não exige a presença de TMOPReifica no projeto do usuário.

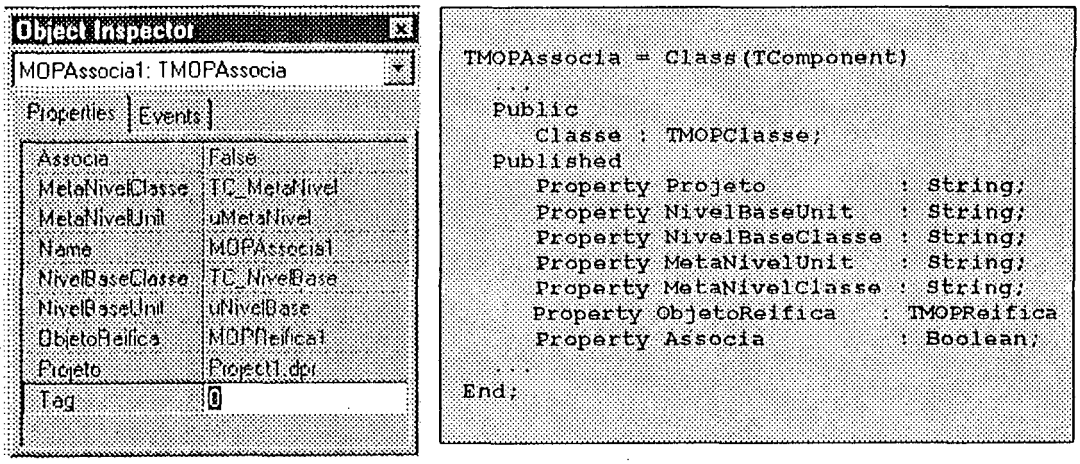
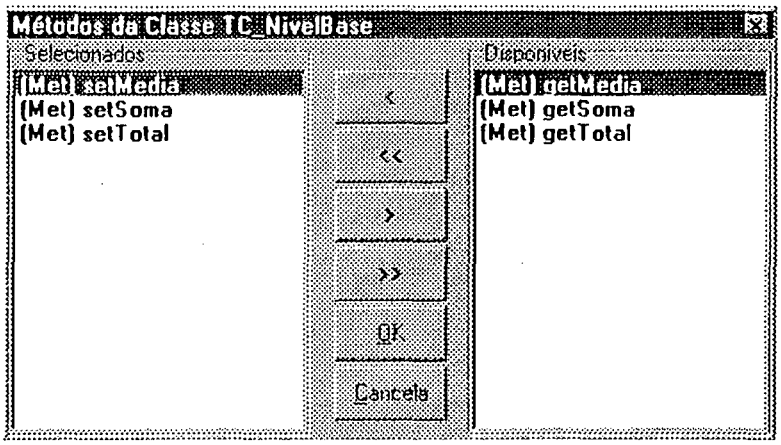


FIGURA 5.10. O COMPONENTE TMOPASSOCIA

5.2.3 O componente TMOPExecuta

O componente TMOPExecuta está baseado no Componente Executa (CE), possuindo a função de permitir a definição de quais métodos serão reflexivos. Daqueles componentes criados, é o único que não está disponível na palheta do ambiente. Sua interface torna-se visível apenas quando forem criados um TMOPExecuta a cada método da classe base reificada. Na figura 5.11, apresenta-se uma janela que surge no momento de um duplo clique sobre TMOPReifica ou TMOPAssocia.



À esquerda, estão os métodos escolhidos de Disponíveis, que, por sua vez, apresenta apenas os de `TC_NivelBase` ainda não selecionados. No botão OK, será instanciado um TMOPExecuta a cada método selecionado e, no Cancela, serão destruídos.

FIGURA 5.11: CRIAÇÃO DE UM TMOPEXECUTA A CADA MÉTODO

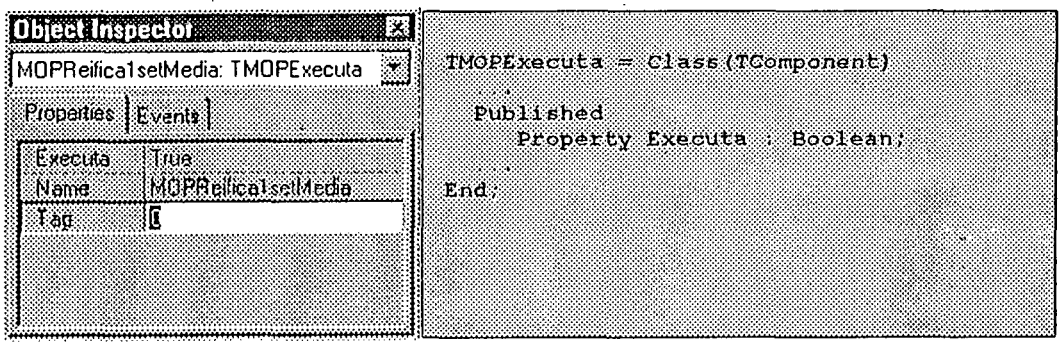


FIGURA 5.12. O COMPONENTE TMOPEXECUTA

A figura 5.12 apresenta a interface de um componente TMOPExecuta. Ao ser ativado o botão OK do formulário da figura 5.11, cada método selecionado é uma instância de TMOPExecuta. Então, os novos componentes criados são:

```
MOPReificaISetMedia : TMOPExecuta;
MOPReificaISetSoma : TMOPExecuta;
MOPReificaISetTotal : TMOPExecuta;
```

O nome de cada instância é composto pelo nome do componente que as criou (neste caso foi MOPReificaI), acrescido pelo nome do método.

5.2.4 Os componentes do grupo *String*

Baseados na proposta de Componente Texto (CT), permitem visualizar uma única meta-informação da classe base reificada. Vários são os componentes visuais que fazem parte deste grupo: TMOPLabel, TMOPButton, TMOPPanel, TMOPBitBtn, TMOPCheckBox, TMOPRadioButton, TMOPGroupBox, e TMOPSpeedButton (os nomes indicam a classe ancestral: TMOPLabel herda TLabel, TMOPButton de TButton). Ao contrário dos componentes não visuais anteriores, que são derivados do TComponent, os do grupo *string* são herdados daqueles que possuem a propriedade Caption. Desta forma, o princípio de funcionamento é atribuir a meta-informação à propriedade Caption, herdada de seus respectivos ancestrais.

Nestes novos componentes criados, observa-se a presença de propriedades comuns. A figura 5.13 detalha apenas um deles e a figura 5.14 apresenta um *form* com vários componentes do grupo *string*.

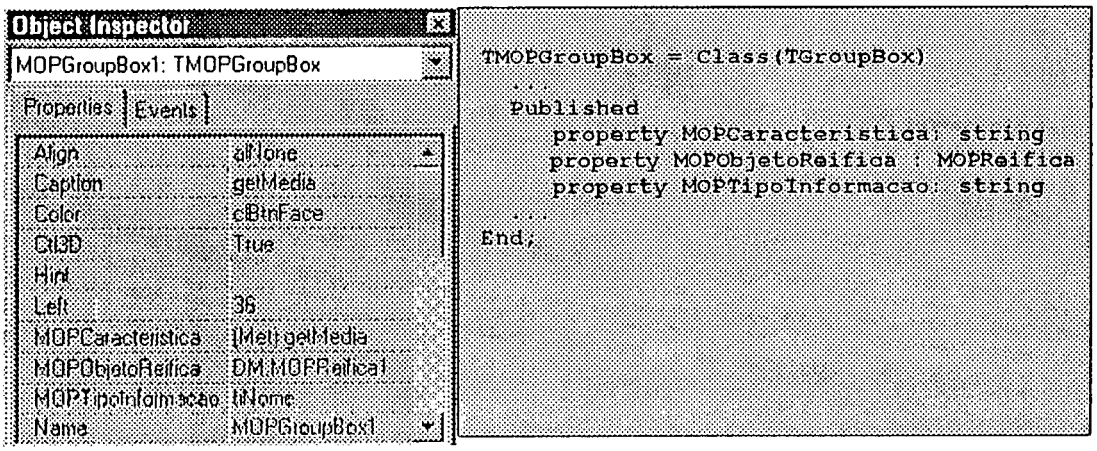
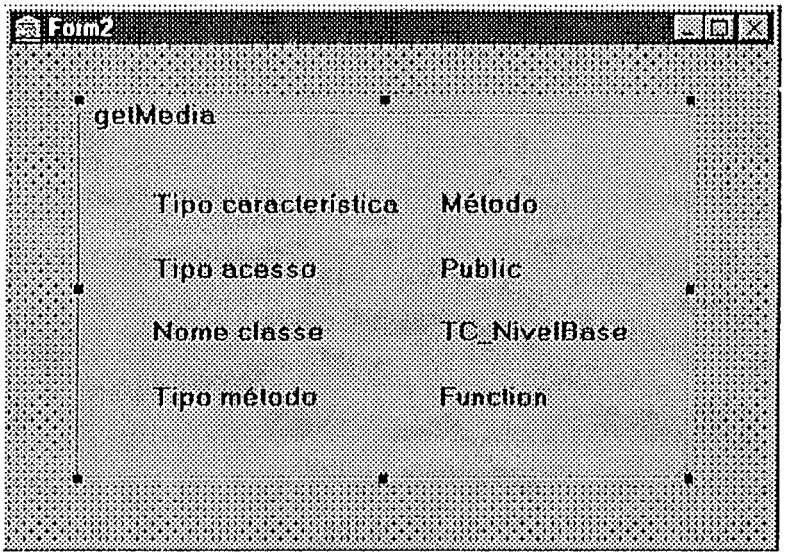


FIGURA 5.13. UM COMPONENTE DO GRUPO *STRING*



O componente TMOPGroupBox tem um Caption `getMedia`. Na coluna da esquerda estão TLabel comuns do Delphi, enquanto que, na direita, vários TMOPLabel.

FIGURA 5.14. UM *FORM* COM VÁRIOS COMPONENTES DO GRUPO *STRING*

A interface de TMOPGroupBox, e também dos demais componentes do grupo *string*, acrescenta três propriedades às herdadas de seu ancestral: MOPObjetoReifica, que deve ser preenchida com uma instância de TMOPReifica a partir de uma lista apresentada; MOPCaracteristica, com a característica da classe base reificada por TMOPReifica; e MOPTipoInformacao, com o tipo de informação da característica escolhida. Os tipos de informação disponíveis estão contidas no escalar TE_TipoInfo, descrito na figura 5.1.

5.2.5 Os componentes do grupo *Lines*

Também baseados na proposta de Componente Texto (CT), permitem visualizar uma lista de meta-informações da classe base reificada. Fazem parte deste grupo TMOPMemo, TMOPListBox, TMOPComboBox e TMOPRadioGroup. Diferentemente dos componentes do grupo *string*, nos quais os ancestrais tinham em comum a propriedade Caption, os do grupo *lines* derivam de componentes que possuem Lines ou Items como propriedade comum (Lines e Items são, na realidade, TStrings, uma classe abstrata do Object Pascal para o tratamento de lista de *strings*).

As figuras 5.15 e 5.16 apresentam, respectivamente, detalhes de apenas um componente do grupo *lines* e um *form* com vários deles.

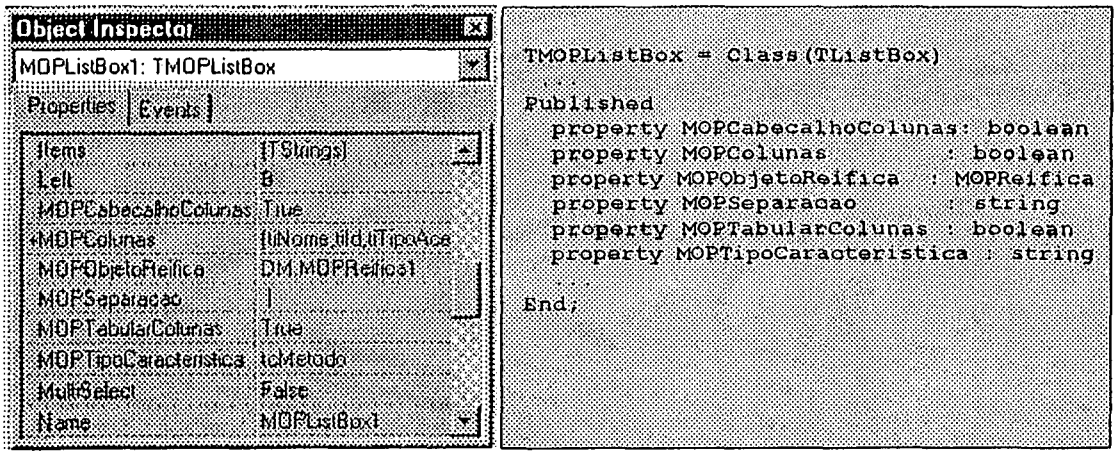


FIGURA 5.15. UM COMPONENTE DO GRUPO *LINES*

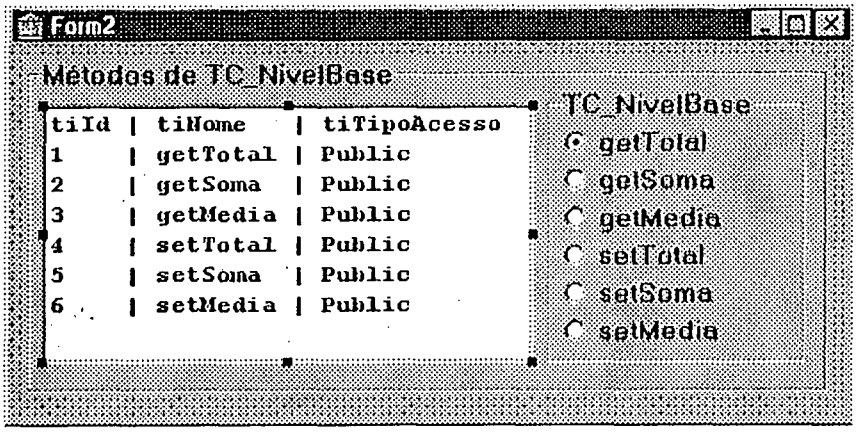


FIGURA 5.16. UM *FORM* COM VÁRIOS COMPONENTES DO GRUPO *LINES*

A interface de TMOPListBox, e também dos demais componentes do grupo *lines*, acrescenta seis propriedades às herdadas de seu ancestral: MOPCabecalhoColuna, que tem a função de habilitar/desabilitar um cabeçalho às colunas na lista resultante de *strings* apresentada pelo componente; MOPColunas, um *set* de TE_TipoInfo, da figura 5.1, que permite que sejam escolhidas as colunas integrantes da lista; MOPObjetoReifica, uma instância de TMOPReifica; MOPSeparacao, uma *string* utilizada na separação das colunas da lista resultante; MOPTabularColunas, que deixa as colunas tabuladas (preenchidas com espaços); e MOPTipoCaracteristica, um escalar

TE_TipoCarac, conforme figura 5.1, para que seja escolhido o tipo de característica, métodos ou atributos, das colunas da lista resultante.

5.3. Aplicações que usam *OP*MOP

Várias aplicações de exemplos foram escritas no intuito de demonstrar a utilização dos componentes *OP*MOP, que, neste momento, serão apenas apresentadas. No entanto, o anexo D descreve-as com maiores detalhes.

Internamente, os exemplos estão padronizados em termos de nomes de *units* e classes. Segue a lista:

- **o projeto**: será sempre `Project1`;
- **a classe base**: está na *unit* `uNivelBase` e tem o nome de `TC_NivelBase`;
- **a interface**: está numa *unit* `uInterface` e contém um *form* chamado `Form1`;
- **a metaclasse**: está em `uMetaNivel` e tem o nome de `TC_MetaNivel`;
- **os componentes não visuais**: estão na *unit* `uDataModule`, que contém um *DataModule* chamado `DM`;
- **os componentes visuais:** estão no próprio `Form1`.

A seguir, as apresentações dos exemplos.

5.3.1. Aplicação 1 – Depurador

A interface é dividida em duas partes: em uma, estão os componentes normais da aplicação do usuário, que invocam métodos da classe base; em outra, uma lista com os métodos executados (figura 5.17). O exemplo atua como um depurador, sendo possível, inclusive, controlar a invocação dos métodos. Os componentes *OP*MOP utilizados foram o TMOPReifica e o TMOPAssocia. A metaclasse atua no sentido de invocar (ou não) os métodos da classe base.

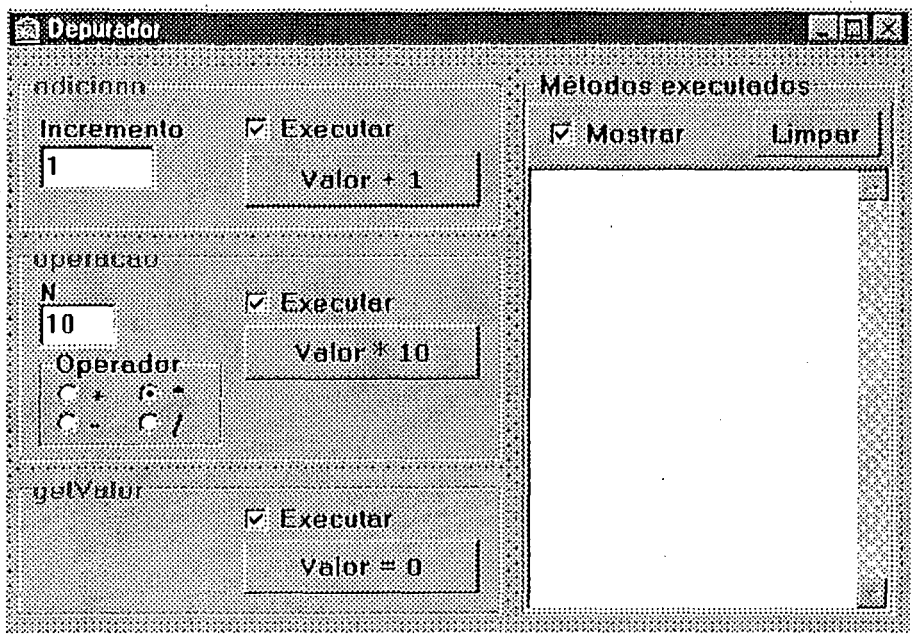


FIGURA 5.17: INTERFACE DO EXEMPLO "DEPURADOR"

5.3.2. Aplicação 2 – Desempenho

Outro exemplo relacionado à interceptação de mensagens. A interface da figura 5.18 apresenta os resultados finais do desempenho das invocações dos métodos da primeira coluna. Foram utilizados TMOPReifica e TMOPAssocia, de *OP*MOP. A função da metaclasse, aqui, é interceptar as mensagens, obter os horários inicial e final de cada invocação e calcular o tempo decorrido.

5.3.3. Aplicação 3 – Meta-informações em *designtime*

As figuras 5.14 e 5.16 apresentam *forms* com vários componentes visuais *OP*MOP. O exemplo que os gerou não tem uma metaclasse definida.

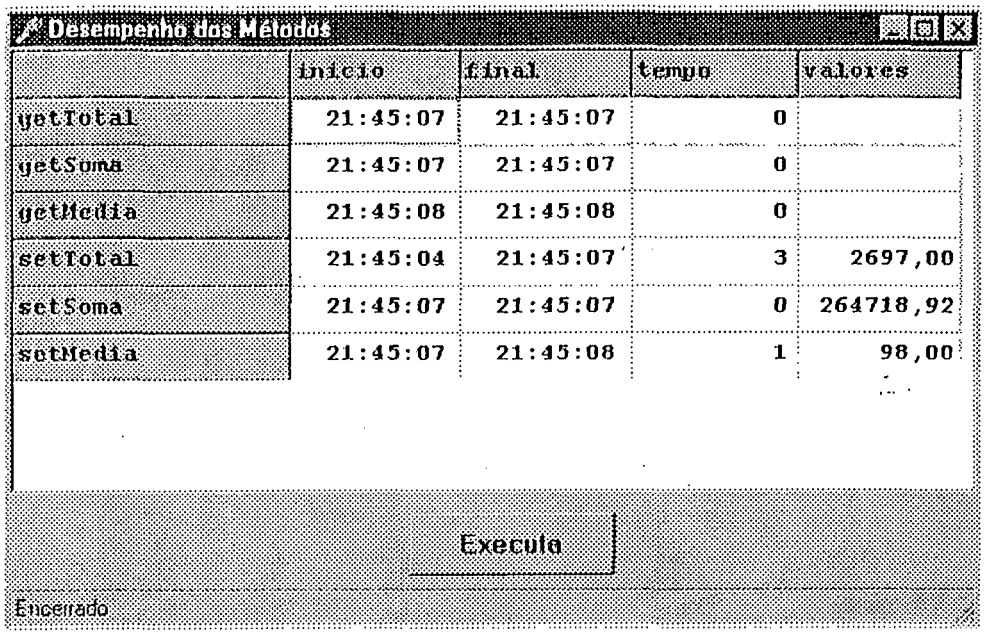


FIGURA 5.18: INTERFACE DO EXEMPLO "DESEMPENHO"

5.3.4. Aplicação 4 – Meta-informações em *runtime*

Ao contrário do exemplo anterior, em que o resultado da reificação é mostrado em tempo de projeto, este, da aplicação 4, mostra como as meta-informações da classe base são obtidas em tempo de execução. A figura 5.19 mostra um *form* com páginas para as informações da classe, dos atributos, dos métodos e dos argumentos. Também não foi criada nenhuma metaclasse.

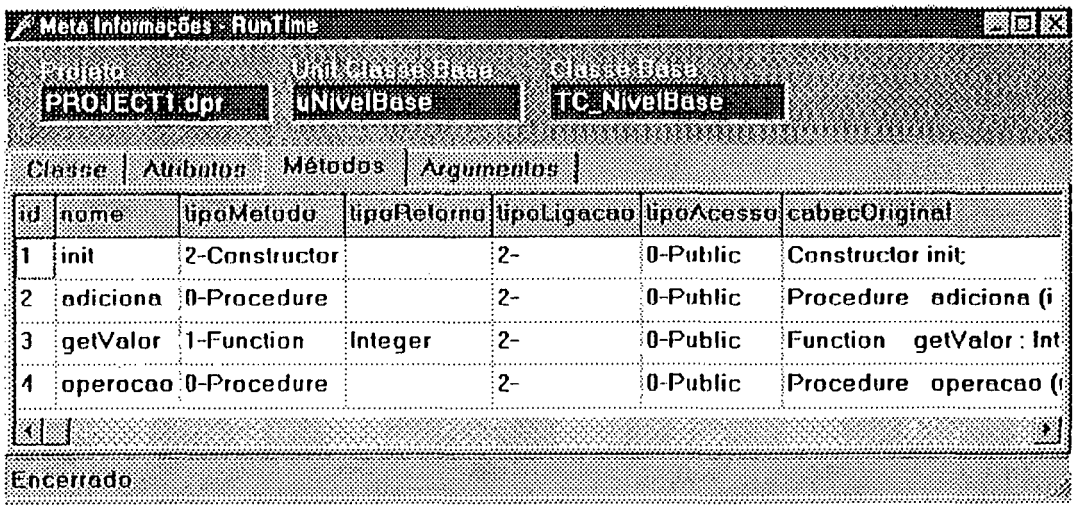


FIGURA 5.19: INTERFACE DO EXEMPLO "META-INFORMAÇÕES EM *RUNTIME*"

Capítulo 6
CONSIDERAÇÕES FINAIS

Este trabalho apresentou, além de uma proposta de suporte à reflexão computacional para ambientes de desenvolvimento visual de software, uma implementação para o Delphi, denominada *OP*MOP. O modelo, voltado à reflexão comportamental, levou em conta o fato de que estes ambientes, por mais aberturas que promovam, não permitem interferências na forma como trabalham e que extensões de funcionalidades às suas linguagens de programação sejam feitas por intermédio de componentes.

Uma primeira avaliação desta proposta deu-se pela implementação *OP*MOP. Pode-se dizer que os resultados foram os esperados e somente não foram melhores devido a limitações de projeto (esta primeira versão do pré-processador não está tratando arquivos DLL e declarações "*include*") e, principalmente, as impostas pelo ambiente Delphi, tais como:

a) não está documentada, no ambiente, uma forma de acesso aos arquivos de um projeto diretamente da memória. A solução adotada de acessá-los do disco exige que a aplicação esteja gravada antes da reificação;
b) não é possível, na atual VCL, um componente saber, no momento em que é instanciado, a *unit* que o contém. Isto forçou a decisão de ser acrescentada, em alguns componentes, uma propriedade na qual deve ser informado o nome do projeto que contém a classe base e a metaclasse;
c) os componentes visuais não conseguem "perceber" atualizações da classe base. Os resultados da reificação são notados visualmente apenas após a invocação explícita desta operação.

Apesar disto, vários são os benefícios desta proposta de suporte:

1. a visualização dos resultados da reificação ainda em tempo de *design* promove uma maior agilidade na definição do algoritmo da metaclasse (a definição da classe base no seu formato original pode não estar disponível);
2. a utilização de componentes permite facilidades no sentido de alternar a situação de aplicações no controle da execução. Em outras palavras, uma aplicação volta a ser não reflexiva com a mesma facilidade com que se tornou reflexiva;
3. novas funcionalidades podem ser implementadas nos componentes através de especializações;
4. as adaptações necessárias à interceptação de mensagens atingem um bom grau de transparência, mesmo alterando diretamente uma classe definida pelo usuário;
5. a característica de criar um componente a cada método da classe base reificada permite alterações incrementais de uma forma clara e objetiva;
6. os constantes ajustes na interface da aplicação em termos de apresentação e comportamento de componentes, bastante comuns nos ambientes visuais de programação, não afetam a estrutura definida para o controle da invocação dos métodos.

Outra questão que preocupa os projetistas de MOP refere-se ao desempenho de uma aplicação que contém um mecanismo de interceptação de mensagens. O apêndice C mostra um estudo comparativo sobre a implementação *OP*MOP. Nos testes aplicados, observou-se que o desempenho foi pouco afetado, pois foram criados, para o provimento da interceptação, objetos com um mesmo modelo, já que são escritos em Object Pascal. Além do mais, as operações de mudança no fluxo de controle entre os objetos são rápidas, por serem feitas por ponteiros na memória. Estes resultados, no entanto, não devem ser considerados conclusivos. Uma análise mais profunda faz-se necessária para a obtenção de resultados mais precisos e confiáveis.

Os componentes visuais *OP*MOP, criados com o intuito de proverem o resultado da pré-compilação em tempo de *design*, foram divididos em dois grupos: os que tratam as meta-informações da classe base como uma única *string* (utilizando a propriedade Caption herdada) e aqueles que as tratam numa lista de *string* (propriedade Lines ou

Items). Outra motivação para utilizá-los, além daquela para a qual foram criados, está na facilidade de preenchimento das propriedades de lista de *strings*, especialmente se a classe base contém uma quantidade considerável de atributos e métodos.

Com base nos resultados obtidos, acredita-se que o trabalho deva ser continuado. A seguir, algumas perspectivas de continuidade:

- implementar o modelo atual noutro ambiente;
- reescrever a proposta de suporte como um *framework* e criar componentes baseados neste *framework*;
- acrescentar à proposta o tratamento de persistência de objetos;
- implementar novos componentes no *OP*MOP, como o tratamento da impressão (semelhante aos da pasta QuickReport) e ao estilo de TStringGrid e de TtreeView.

REFERÊNCIAS BIBLIOGRÁFICAS

[ADO00] ADOLPH, Helmuth. **ProDelphi User Guide Release 8.3.** Documento WWW capturado de http://www.prodelphi.de em outubro de 2000.

[ALV97] ALVES, William Pereira. **Delphi 3. Programação Visual para Windows**. São Paulo: Érica, 1997.

[BEK93] BEKKER, Carel; HEEVER, Roelf Vab Den. Albedo, a metaobject infrastructure for Smalltalk. In: OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES AND APPLICATIONS; WORKSHOP ON OO REFLECTION AND META-LEVEL ARCHITECTURES, 1993. **Proceedings...,** 1993.

[BRE92] BRETTHAUER, Harry; DAVIS, Harley; KOPP, Jürgen; PLAYFORD, Keith. Balancing the EuLisp Metaobject Protocol. In IMSA'92 INTERNATIONAL WORKSHOP ON REFLECTION AND META-LEVEL ARCHITECTURE. Tokyo, 1992. **Proceedings...,** 1992.

[BOB88] BOBROW, Daniel; DIMICHIEL, L.G.; GABRIEL, R.P.; KEENE, S.E.; KICZALES, Gregor; MOON, D.A. A Common Lisp Object System Specifications. In: **SIGPLAN Notices**, vol. 23, 1988.

[BOS97] BOSCH, Jan. Adapting Object-Oriented Components. SECOND INTERNATIONAL WORKSHOP ON COMPONENT-ORIENTED PROGRAMMING (1997) – Jyväskylä, Finlândia, 1997. **Proceedings...,** 1997

[CAM97] CAMPO, Marcelo Ricardo. **Compreensão Visual de Frameworks através da Introspeção de Exemplos**. Tese de doutoramento, programa de pós-graduação em Ciência da Computação (CPGCC), Universidade Federal do Rio Grande do Sul (UFRGS), Porto Alegre, março 1997.

[CAN96] CANTÚ, Marco. **Dominando o Delphi 2**. São Paulo: Makron Books, 1996.

[CHI95] CHIBA, Shigeru. A Metaobject Protocol for C++. In: **ACM SIGPLAN Notices**, v. 30. Trabalho apresentado na OOPSLA, 1995

[CHI96] CHIBA, Shigeru. Open C++ Programmer's Guide for Version 2. **Xerox PARC**. Technical Report, SPL-96-024, 1996.

[CHI97] CHIBA, Shigeru. **Open C++ 2.5 Reference Manual**. Institute of Information Science and Eletronics. University of Tsukuba. 1997.

[CHI98a] CHIBA, Shigeru. MICHIAKI, Tatsubori. **Open Java Tutorial**. Institute of Information Science and Eletronics. University of Tsukuba, 1998.

[CHI98b] CHIBA, Shigeru. Javassist – A Reflection-based Programming Wizard for Java. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING – ECOOP - WORKSHOP ON REFLECTIVE OBJECT-ORIENTED PROGRAMMING AND SYSTEMS, julho 1998. **Proceedings...**, 1998

[DAV97] DAVIS, Harold. **Delphi: Ferramentas Poderosas.** Tradução Ana Beatriz Tavares dos Santos Pereira. São Paulo: Berkeley Brasil, 1997.

[ENT93] ENTSMINGER, Gary. **Segredos dos Mestres do Visual Basic 3 for Windows.** Tradução Marcelo Vieira de Brito. Rio de Janeiro: Berkeley Brasil, 1993.

[FOO93] FOOTE, Brian. Architectural Balkanozation in the Post-Linguistic Era. In: OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES AND APPLICATIONS - OOPSLA - WORKSHOP ON OBJECT-ORIENTED REFLECTION AND META-LEVEL ARCHITECTURES, 1993. **Proceedings...**, 1993

[FUR98] FURLAN, José Davi. **Modelagem de Objetos através da UML** – the Unified Modeling Language. São Paulo: Makron Books, 1998.

[GRA89] GRAUBE, N. Metaclass compatibility. In: **ACM SIGPLAN Notices**, v.24. Trabalho apresentado na OOPSLA, 1989.

[HIL92] HILS, Daniel David. **A Visual Programming Language for Visualization of Scientific Data.** Master Thesis. University of Illinois, Urbana, 1992.

[ICH92] ICHISUGI, Yuuji; MATSUOKA, Satoshi;YONEZAWA, Akinori. RbCl: A Reflective Object-Oriented Concurrent Language without a Run-Time Kernel. In IMSA'92 INTERNATIONAL WORKSHOP ON REFLECTION AND META-LEVEL ARCHITECTURE. Tokyo, novembro 1992. **Proceedings...**, 1992.

[JOH88] JOHNSON, Ralph; FOOTE, Brian. Designing Reasable Classes. **Journal of Object-Oriented Programming**, vol 1, no. 2, 1988.

[JOH89] JOHNSON, Ralph; FOOTE, Brian. Reflective Facilities in Smalltalk-80. In: **ACM SIGPLAN Notices**, v.24, n.10, 1989.

[JOH91] JOHNSON, Ralph; RUSSO, Vincent. **Reusing Object-Oriented Design**. Technical Report UIUCDCS 91-1696, University of Illinois, maio 1991.

[JOH92] JOHNSON, Ralph. Documenting Frameworks Using Patterns. In:. OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES AND APPLICATIONS, 1992. **Proceedings...**, 1992.

[JON94] JONATHAN, Miguel. Introdução à Programação Orientada a Objetos com Smalltalk. In: JORNADA DE ATUALIZAÇÃO EM INFORMÁTICA, CONGRESSO SBC, 1994. **Anais**, 1994

[KIC91] KICZALES, Gregor. RIVIERES, Jim des; BOBROW, Daniel. Metaobject Protocols. In: ACM CONFERENCE ON OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES AND APPLICATIONS - WORKSHOP ON REFLECTION AND METALEVEL ARCHITECTURES IN OBJECT-ORIENTED PROGRAMMING,1991. **Proceedings...**, 1991.

[KIC97] KICZALES, Gregor. LAMPING, John; MENDHEKAR, Anurang; MAEDA, Chris; LOPES, Cristina; LOINGTIER, Jean-Marc; IRWIN; John. Aspect-Oriented Programming In. EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING – ECOOP - Finland, 1997. **Proceedings...**, 1997

[KIP93] KIPPER, Eti Francisco; MULLER, Carlos Alberto; BASTS, Érico de Almeida; CEVALLOS; Jorge; JAEGER, José Inácio; RESENDE, Marco Antônio. **Engenharia de Informações: conceitos, técnicas e métodos**. Porto Alegre: Sagra-DC Luzzatto, 1993.

[KLE95] KLEYN, Michiel Florian Eugene. **A High Level Language for Specifying Graph-Based Languages and Their Programming Environments**. Master Thesis. University of Texas, 1995.

[KRU94] KRUGLINSKI, David. **Explorando Visual C++**. Tradução Geraldo Costa Filho. Rio de Janeiro: Campus, 1994.

[LIM96] LIMA, Adilson da Silva. **Programando em Visual Basic 4.0, Delphi 2.0 e SQLWindows 5.0**. São Paulo: Érica, 1996.

[LIS97] LISBÔA, Maria Lúcia. Arquiteturas de Meta-nível. In: XX SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE. FORTALEZA, CE, outubro 1997. **Anais**, 1997

[LOP99] LOPES, Cristina Videira; KICZALES, Gregor. **Recent Developments in AspectJ.** Xerox PARC. Documento WWW capturado de http://www.parc.xerox.com/spl/projects/aop/aspectj em dezembro 1999.

[MAE87] MAES, Pattie. **Computational Reflection**. Phd Thesis. Technical Report 87.2, Artificial Intelligence Laboratory, Vrije Universiteit Brussel, 1987.

[MAE88] MAES, Pattie. Issues in Computational Reflection. In WORKSHOP META-LEVEL ARCHITECTURES AND REFLECTION, Italy, 1988. **Proceedings...**, 1988

[MAR95] MARTIN, James; ODELL, James. **Análise e Projeto Orientados a Objeto**. São Paulo: Makron Books, 1995.

[MAT99a] MATTSSON, Michael. **Evolution and Composition of Object-Oriented Frameworks**. Phd Thesis. Technical Report LU-CS-TR:96-167, Department of Software Engineering And Computer Science, University of Karlskrona, Ronneby, Sweden, 1999.

[MAT99b] MATTSSON, Michael; BOSCH, Jan. **Framework Composition: Problems, Causes and Solutions**. Department of Software Engineering And Computer Science, University of Karlskrona, Ronneby, Sweden. Documento WWW capturado de http://www.ide.hk-r.se/~michaelm/ em novembro de 2000.

[MIC97] MICHAIL, Amir. **Visual Programming without Procedures.** Technical Report UW-CSE-97-05-02, University of Washington, maio 1997.

[MOS99] MÖSSEMBÖCK, Hanspeter; STEINDL, Christoph. The Oberon-2: Reflection Model and Its Application. In SECOND INTERNATIONAL CONFERENCE META-LEVEL ARCHITECTURES AND REFLECTION. Springer-Verlag, 1999 (Lecture Notes on Computer Science n. 1616). **Proceedings...**, 1999

[NAK92] NAKAJIMA, Shin. What Makes a Language Reflective and How?. In IMSA'92 INTERNATIONAL WORKSHOP ON REFLECTION AND META-LEVEL ARCHITECTURE. Tokyo, novembro 1992. **Proceedings...**, 1992.

[NIE99] NIELSEN, Henrik; ELMSTROM, René. Proposal for Tools Supporting Component Based Programming. In: FOURTH INTERNATIONAL WORKSHOP ON COMPONENT-ORIENTED PROGRAMMING – WCOP'99 –1999. **Proceedings...**, 1999

[OLI98a] OLIVA, Alexandre; BUZATO, Luiz Eduardo. **Guaraná: A Tutorial.** Technical Report IC-98-31, Instituto de Computação, Universidade Estadual de Campinas, Campinas, SP, setembro 1998

[OLI98b] OLIVA, Alexandre; GARCIA, Islene Calciolari; BUZATO, Luiz Eduardo. **The Reflexive Architecture of Guaraná**. Technical Report IC98-14, Instituto de Computação, Universidade Estadual de Campinas, Campinas, SP, abril 1998

[ORF97] ORFALI, Robert; HARKEY, Dan. **Client/Server Programming with Java and CORBA.** Wiley Computer Publishing, 1997.

[PAC99] PACHECO, Xavier. **A Visual Component Library.** Delphi Developer Support, Borland International Inc. Documento WWW capturado de http://www.inprise.com/articles em agosto de 2000.

[PAI96] PAINTER, Robert Andrew; TURNER, Joseph. **Tutorial VisualWorks 2.0**. Clemson University. Documento WWW capturado de http://www.cs.clemson.edu/~lab428/VW em novembro de 2000.

[PAR94] PARCPLACE SYSTEMS. **VisualWorks User's Guide**. Sunnyvale: ParcPlace Systems, 1994.

[ROB98] ROBERTS, Don; JOHNSON, Ralph. **Evolving Frameworks. A Pattern Language for Developing Object-Oriented Frameworks.** University of Illinois. Documento WWW capturado de http://st-www.cs.uiuc.edu/users/droberts em novembro de 2000.

[ROB99] ROBBEN, Bert; VANHAUTE, Bart; JOOSEN, Wouter; VERBAETEN, Pierre. Non-functional Policies. In SECOND INTERNATIONAL CONFERENCE, META-LEVEL ARCHITECTURES AND REFLECTION. Springer-Verlag, 1999 (Lecture Notes on Computer Science n. 1616). **Proceedings...,** 1999

[RUM91] RUMBAUGH, James; BLAHA, Michael; PREMERLANI, William; EDDY, Frederick; LORENSEN, William.. **Object-Oriented Modeling and Design**. Prentice Hall, New Jersey, 1991.

[SHA95] SHAW, Mary; GARLAN, David. Formulation and Formalisms in Software Architecture. LECTURE NOTES IN COMPUTER SCIENCE n.1000. Berlin, Springer, **Proceedings...,** 1995

[SIL00] SILVA, Ricardo Pereira e. **Suporte ao Desenvolvimento e Uso de Frameworks e Componentes**. Tese de doutoramento, programa de pós-graduação em Ciência da Computação (CPGCC), Universidade Federal do Rio Grande do Sul (UFRGS), Porto Alegre, março 2000.

[SMI82] SMITH, Brian. **Reflection and Semantics um a Procedural Language**. Technical Report TR 272, Laboratory for Computer Science, Massachusetts, Institute of Technology, 1982.

[STE94] STEEL, Luc. Beyond Objects. In. EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING – ECOOP - Italia, 1994. (Lecture Notes in Computer Science n. 821) Springer-Verlag, **Proceedings...,** 1994.

[STE96] STEYAERT, Patrick; HONDT, Koean; DEMEYER, Serge; BOYEN, Niels. Reflective Application Builders. In: INTERNACIONAL CONFERENCE ON OBJECT-ORIENTED INFORMATION SYSTEMS. Springer-Verlag, janeiro 1996. **Proceedings...,** 1996.

[SZY96] SZYPERSKI, Clemens; PFISTER, Cuno. In FIRST INTERNATIONAL WORKSHOP ON COMPONENT-ORIENTED PROGRAMMING – WCOP'96 – Linz, Austria, 1996. **Report...,** 1996

[SZY99] SZYPERSKI, Clemens. **Component Software. Beyond Object-Oriented Programming**. Addison-Wesley, 1999.

[SWA92] SWAN, Tom. **Programando em Turbo Pascal for Windows 3.0**. Vol.I, Rio de Janeiro: Berkeley, 1992.

[WEC97] WECK, Wolfgang; BOSCH, Jan; SZYPERSKI, Clemens; In: SECOND INTERNATIONAL WORKSHOP ON COMPONENT-ORIENTED PROGRAMMING – WCOP'97 – Jyväskylä, Finlândia, 1997. **Resume...**, 1997

[WEL98] WELCH, Ian; STROUD, Robert. Dalang – A Reflective Java Extension. In OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES AND APPLICATIONS - OOPLSA - WORKSHOP ON REFLECTIVE PROGRAMMING IN C++ AND JAVA, 1998. **Proceedings...**, 1998.

[WEL99] WELCH, Ian; STROUD, Robert. From Dalang to Kava – the Evolution of a Reflective Java Extension. In SECOND INTERNATIONAL CONFERENCE, META-LEVEL ARCHITECTURES AND REFLECTION. Springer-Verlag, 1999 (Lecture Notes on Computer Science n. 1616). **Proceedings...**, 1999

[ZAN97] ZANCANELLA, Luiz Carlos. **Estrutura Reflexiva para Sistemas Operacionais Multiprocessados.** Tese de doutoramento, programa de pós-graduação em Ciência da Computação (CPGCC), Universidade Federal do Rio Grande do Sul (UFRGS), Porto Alegre, dezembro de 1997.

[ZIM96] ZIMMERMANN, Chris. **Advances in Object-Oriented Metalevel Architectures and Reflection**. Berkeley CRC Press, 1996.

APÊNDICE A
UMA VISÃO DO DELPHI

Lançado pela Borland International Inc. em 1994 para competir com Visual Basic, o Delphi rapidamente tornou-se uma ferramenta bastante popular para o desenvolvimento de aplicações no Windows. Ele é baseado no Object Pascal, uma linguagem híbrida que combina o Pascal procedimental com extensões da POO.

A.1. Breve histórico

O Pascal foi desenvolvido por Nicklaus Wirth em 1960 como resposta a outra linguagem desenvolvida na época, o Algol, para incorporar uma abordagem oposta a este, sendo menor e possibilitando combinar poder e simplicidade. A idéia chave da nova linguagem era ordem, administrada por meio de um forte conceito de tipo de dados. Na verdade, o Pascal foi e continua sendo usado como uma das primeiras linguagens de programação que estudantes de computação aprendem.

Originalmente, o Pascal foi visto como uma linguagem acadêmica, até o lançamento, em 1984, do Turbo Pascal, da Borland. Embora existissem outras implementações do Pascal para micros, a primeira versão do Turbo Pascal era tecnologicamente superior aos outros, especialmente no desempenho da compilação, originando daí o nome de Turbo. A Borland continuou trabalhando no produto, alterando a arquitetura básica da linguagem na versão 3 com a introdução do conceito de *units* e, na versão 5.5, com o conceito de objetos. Em 1992, surgiu a última versão para o ambiente MS-DOS, denominada de 7, juntamente com o lançamento do Turbo Pascal for Windows, para o Windows 3.x, que utilizava a biblioteca OWL (*Object Windows Library*). A primeira versão do Delphi veio em 1994, ainda para o Windows 3.x, baseada no Object Pascal e já utilizava a VCL (*Visual Component Library*) para o

desenho das interfaces. A versão 2.0 foi lançada logo em seguida para o Windows 95. Atualmente está na versão 5.0.

A.2. Características

Delphi é uma ferramenta de desenvolvimento visual para criação de aplicações Windows, apresentado num ambiente integrado de editor de textos, compilador e depurador.

Algumas características são:

- suporte a banco de dados, incluindo Oracle, Sybase e Informix;
- compilador muito rápido;
- abordagem baseada em formulários e orientada a objetos;
- uma biblioteca de componentes nativos e de terceiros;
- possibilidade do uso de ferramentas de terceiros.

Para a criação da interface da aplicações, o usuário dispõe de uma biblioteca de componentes, denominada **VCL** (*Visual Component Library*). Ela é composta de classes não só relacionadas à interface como também para fins gerais, além de conter classes que não são componentes. A figura 3.4 mostra a palheta de componentes do Delphi. Cada página agrupa componentes de acordo com suas funcionalidades ou então oriundos de um mesmo fornecedor. Desta forma, a palheta abriga não somente os componentes padrão do Delphi, mas também os novos componentes desenvolvidos por terceiros.

A hierarquia de classes do Delphi introduzida já na versão 1.0 é, na realidade, um *framework* utilizado para a construção das aplicações manipuladas em tempo de projeto. Isto permite que alterações no comportamento e características visuais dos componentes sejam percebidas antes mesmo da execução da aplicação.

A.3. O ambiente

É composto de quatro janelas principais, mostradas na figura 3.1. São elas:

- principal: contém um menu com opções suspensas, botões de atalho para as tarefas mais rotineiras (os SpeedButtons) e a palheta de componentes. Ela é denominada de principal porque provoca o encerramento da execução do Delphi quando for fechada.
- *Object Inspector*: permite inspecionar os objetos do *form*, mostrando suas propriedades e os eventos associados a cada um (figura 3.7).
- *form*: é o formulário com os componentes gráficos da interface. É no *form* que o programador desenhará sua interface.
- *unit*: um arquivo texto que contém ou o código associado ao *form* ou código Object Pascal com rotinas genéricas escritas pelo programador da aplicação.

A.4. Escrevendo uma aplicação

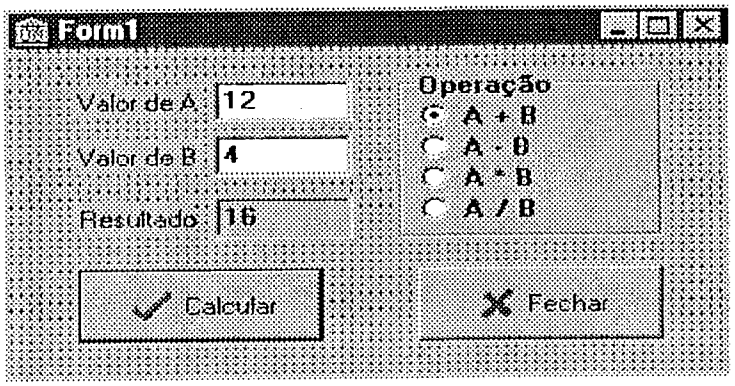
O desenvolvimento de uma aplicação Delphi tem início pela criação de um ***Project***. Um *project* é uma coleção dos arquivos fontes em Object Pascal que comporão ou uma aplicação ou uma DLL. Alguns destes arquivos são gerados em tempo de projeto, enquanto outros, pelo compilador. A figura A.1 mostra um exemplo de um projeto chamado `Project1`.

Os arquivos fontes em Object Pascal que comporão um projeto são colocados em ***units***. Desta forma, pode-se dizer ainda que um projeto é uma coleção de *units* Delphi. Uma *unit* é uma biblioteca que contém declarações e rotinas úteis à aplicação. Este mecanismo agrupa classes, funções, procedimentos, tipos, constantes e variáveis, e permite ainda o ocultamento de informações. Pelo fato do Delphi prover facilidades no desenho de interfaces, os componentes visuais (e não visuais também) da interface da aplicação ficam armazenados em um ***form***. Associada a este *form* existe uma *unit* com o código fonte dos eventos definidos para estes objetos. Uma *unit* não necessariamente está associada a um *form*, mas um *form* sempre está associado a uma *unit*. As figuras A.2, A.3 e A.4 mostram exemplos de *form* e de *units*.

```
<1>  program Project1;
<2>
<3>  uses  Forms,
<4>        Unit1 in 'Unit1.pas',
<5>        Unit2 in 'Unit2.pas' {Form1},
<6>        Unit3 in 'Unit3.pas';
<7>
<8>  {$R *.RES}
<9>
<10> begin
<11>       Application.Initialize;
<12>       Application.CreateForm(TForm1, Form1);
<13>       Application.Run;
<14> end.
```

Os números envolvidos por <> não fazem parte do fonte, servem apenas para indicação do número da linha. As palavras em negrito são palavras reservadas da linguagem. Em <3> tem uma cláusula **uses**. Ela indica os arquivos que fazem parte do *project*, ou seja, as *units*. Em <8> está um arquivo mantido pelo Delphi e entre <10> e <14> a execução da aplicação.

FIGURA A.1: UM ARQUIVO *PROJECT* DO DELPHI



Este Form1 representa a interface da aplicação do usuário. A linha <5> da figura A1 indica que ele faz parte do projeto Project1

FIGURA A.2: UM *FORM* EM DELPHI

```
<1> object Form1: TForm1
<2>   Left = 220
<3>   Top = 137
<4>   Width = 304
<5>   Height = 196
<6>   Caption = 'Form1'
<7>   object Label1: TLabel
<8>     Left = 28
<9>     Top = 20
<10>    Width = 49
<11>    Height = 13
<12>    Caption = 'Valor de A'
<13>  end

<106> object BitBtn2: TBitBtn
<107>   Left = 168
<108>   Top = 112
<109>   Width = 101
<110>   Height = 37
<111>   Caption = 'Fechar'
<112>   TabOrder = 2
<113>   OnClick = BitBtn2Click
<114>   Kind = bkCancel
<115> end
<116> end
```

Parte de um programa fonte controlado pelo Delphi nos bastidores do projeto do usuário. Está associado ao *form* Form1. Toda e qualquer alteração em Form1 é registrada neste fonte. No exemplo o usuário foi poupado da digitação de 116 linhas de código Object Pascal

FIGURA A.3: BASTIDORES DO PROJETO DE UM *FORM*

A linha <3> da figura A.4 mostra a palavra reservada **interface**. Ela determina que toda declaração presente nesta seção será visualizada pelas outras *units*. A linha <5> tem **uses** seguida de uma lista com todas as *units* que farão parte de `Unit2`. Notar que **uses** não é uma declaração "*include*". Na verdade, serão incorporadas ao código de `Unit2` apenas as declarações presentes na seção da *interface* de cada *unit* da lista de **uses**. A linha <9> mostra a declaração de uma classe chamada `TForm1` que herda `TForm`, da estrutura de classes do Object Pascal. Esta linha, juntamente com as <10> a <20>, são atualizadas pelo Delphi a cada alteração no *form*

Uma outra seção de uma *unit* é a **implementation**, na linha <30>. É nela que serão colocados os algoritmos da aplicação. Toda declaração de identificador nela contida será visível apenas nesta *unit*, não sendo compartilhada por outras. A linha <32> mostra a inclusão do arquivo fonte dos bastidores, conforme figura A.3.

```
<1>  unit Unit2;
<2>
<3>  interface
<4>
<5>  uses Windows, Messages, SysUtils, Classes, Graphics, Controls,
<6>       Forms, Dialogs, StdCtrls, ExtCtrls, Buttons;
<7>
<8>  type
<9>    TForm1 = class(TForm)
<10>       BitBtn1: TBitBtn;
<11>       BitBtn2: TBitBtn;
<12>       Label1: TLabel;
<13>       Edit1: TEdit;
<14>       Label2: TLabel;
<15>       Edit2: TEdit;
<16>       RadioGroup1: TRadioGroup;
<17>       Label3: TLabel;
<18>       Edit3: TEdit;
<19>       procedure BitBtn2Click(Sender: TObject);
<20>       procedure BitBtn1Click(Sender: TObject);
<21>   private
<22>       { Private declarations }
<23>   public
<24>       { Public declarations }
<25> end;
<26>
<27> var
<28>   Form1: TForm1;
<29>
<30> implementation
<31>
<32> {$R *.DFM}
<33>
<34> procedure TForm1.BitBtn2Click(Sender: TObject);
<35> begin
<36>    Form1.Close;
<37> end;
<38>
<39> procedure TForm1.BitBtn1Click(Sender: TObject);
<40> var A, B, Resultado : Real;
<41> begin
<42>    A := StrToFloat(Form1.Edit1.Text);
<43>    B := StrToFloat(Form1.Edit2.Text);
<44>    Case Form1.RadioGroup1.ItemIndex of
<45>          0 : Resultado := A + B;
<46>          1 : Resultado := A - B;
<47>          2 : Resultado := A * B;
<48>          3 : Resultado := A / B;
<49>    End;
<50>    Form1.Edit3.Text := FloatToStr(Resultado);
<51> end;
<52>
<53> end.
```

FIGURA A.4 : UMA *UNIT* ASSOCIADA AO UM *FORM*

Conforme já comentado, uma *unit* não necessariamente está associada a um *form*. Ela pode conter implementações de classes, procedimentos e funções integrantes da aplicação do usuário. Esta flexibilidade de se trabalhar com *units* pode, infelizmente, levar a um estilo confuso de programação. Uma boa técnica, mas não uma regra, é utilizar uma *unit* a cada funcionalidade da aplicação, ou seja, não "reaproveitar" arquivos. Exemplos:

(a) escrever apenas uma classe dentro de um arquivo de *unit*. Se a aplicação exigir três classes, colocá-las em arquivos distintos;

(b) na *unit* de um *form* deixar apenas o código referente às questões visuais;

(c) colocar todas as funções e os procedimentos desenvolvidos dentro de um único arquivo de *unit*.

Em relação ao arquivo de projeto de uma aplicação, a orientação é deixar que apenas o Delphi mantenha-o atualizado. O ambiente possui facilidades para o gerenciamento de projetos que dispensam a intervenção manual do programador. Em outras palavras, a orientação é deixar que o Delphi controle este arquivo e, se possível, ao escrever uma aplicação, não abri-lo.

A.5. *Form* e *DataModule*

A figura A.2 mostra um *form* contendo alguns componentes visuais. Nele pode-se observar que "Valor de A" é um componente do tipo TLabel em que sua propriedade Caption está definida como "Valor de A". Vale o mesmo para "Valor de B" e "Resultado". A caixa branca com um número 12 é outro componente mas do tipo TEdit, o qual permite que dados sejam editados, isto é, uma caixa de entrada de dados. As caixas com 4 e 16 também são do tipo TEdit. Os botões são do tipo TButton, em que a propriedade Caption de cada um possui valores diferentes. Em suma, um *form* pode conter componentes visuais e não visuais. Formalmente falando, um componente é dito visual quando possui a característica de ser visto, ainda em tempo de desenvolvimento, na forma como se apresentará em tempo de execução, e é dito não visual quando não possui esta característica (não são eles próprios visíveis em tempo de execução, mas

podem, no entanto, gerenciar algo que é visual, como, por exemplo, um componente para caixa de diálogos).

O Delphi inclui ainda um outro tipo de formulário chamado ***DataModule.*** Um *DataModule* é um tipo especial de formulário que abriga apenas os componentes não visuais. Em tempo de desenvolvimento, apresenta-se como um *container* com vários componentes não visuais, tais como TTable e TDatabase entre outros. Possui as mesmas funcionalidades que um *form*, exceto aquelas ligadas ao aspecto visual.

Basicamente, escrever uma aplicação Delphi é escolher uma série de componentes, colocá-los dentro de um *form* (ou de um *DataModule*) e definir suas interações.

APÊNDICE B

FRAMEWORKS E COMPONENTES

A reutilização de software tem sido uma das principais metas da Engenharia de Software por vários anos. Além de promover uma redução dos custos do desenvolvimento, a reutilização permite também uma maior eficiência da equipe de análise e programação. Esta reutilização, no entanto, refere-se não somente ao código das aplicações, mas também ao projeto destas aplicações. Numa breve comparação, os projetistas da época da orientação a objetos alcançam níveis de reutilização de código maiores do que aqueles obtidos na época das técnicas procedimentais. As bibliotecas de procedimentos e funções e, mais tarde, as bibliotecas de classes, mantinham o foco principal na reutilização do código do software. Mattsson [MAT99a] coloca que, em tese, a reutilização de projeto seria mais vantajosa economicamente que a reutilização de código, porque enquanto o projeto retrata a parte intelectual do software, o código é uma conseqüência deste, sendo, portanto, mais fácil de recriá-lo.

Da exploração maciça dos paradigmas da orientação a objetos, novas idéias aparecem no campo do desenvolvimento do software, não só com o intuito de estender funcionalidades, como também para cobrir falhas e limitações nas abordagens existentes. Como uma conseqüência natural, surgem, então, os *frameworks* e os componentes, comentados a seguir.

B.1. *Frameworks*

São encontradas várias definições de ***frameworks***. Destaca-se aqui a de Ralph Johnson e Brian Foote [JOH88], como sendo um conjunto de classes que personifica um projeto abstrato de soluções para uma família de problemas relacionados. Pode-se também dizer que um *framework* é uma estrutura de classes concretas e abstratas, portanto de implementação inacabada, projetada para um domínio particular (apenas por uma questão de padronização, os termos "problemas relacionados" e "domínio particular" serão tratados como "domínios de aplicação"). Um *framework* surge, então,

como fruto da generalização de um domínio de aplicação. A figura B.1 mostra um processo de identificação de um *framework*. Se for possível observar algumas funcionalidades e estruturas em comum dentre várias aplicações de um mesmo domínio, a interseção destas características pode dar origem a um *framework*.

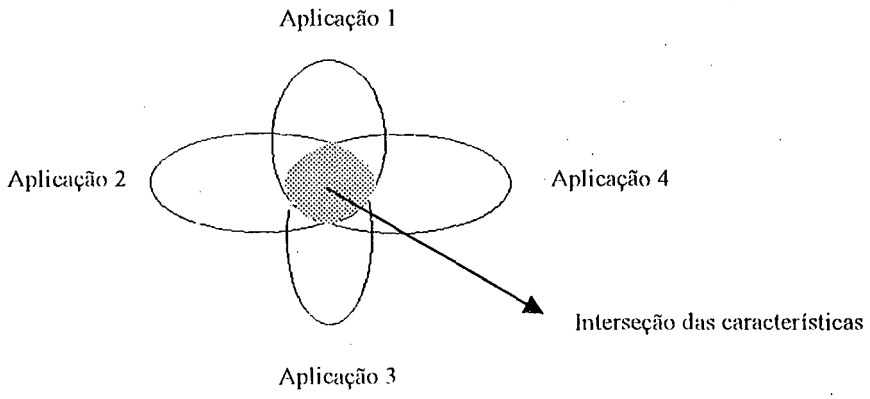
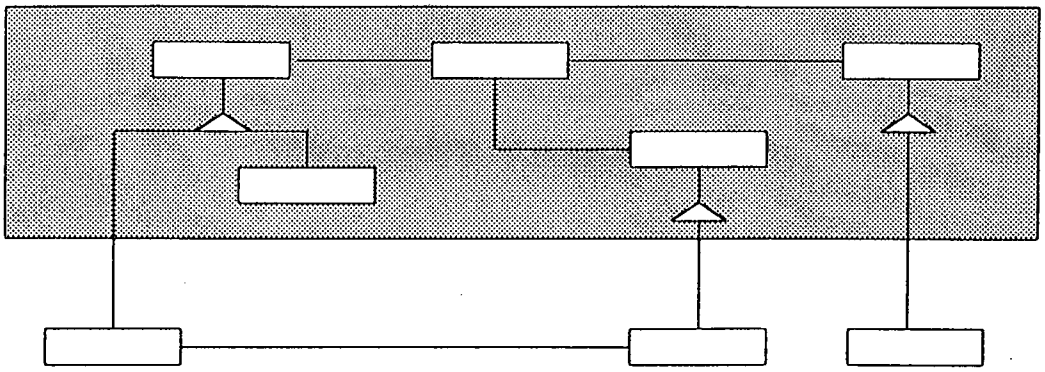


FIGURA B.1: IDENTIFICANDO UM *FRAMEWORK*

Uma vez identificadas, estas funcionalidades e estruturas são retiradas de cada aplicação para formarem o *framework*. Desta forma, tem-se a principal motivação do uso de *framework*, que é a reutilização do software (código e projeto). As aplicações que se utilizarem dele ganham em produtividade devido à reutilização. A figura B.2 apresenta um *framework* genérico e uma aplicação que o usa.

Ambientes de desenvolvimento de software, aplicações de comércio eletrônico na Internet, aplicações para apresentação de resultados científicos e automação industrial e comercial são alguns exemplos de domínios de aplicação.



A parte sombreada apresenta o *framework* como uma estrutura de classes interrelacionadas. Abaixo estão as definições da aplicação que o usam. Por ser uma implementação inacabada, a aplicação precisa especializar algumas classes.

FIGURA B.2: UMA APLICACÃO QUE USA UM *FRAMEWORK*

B.1.1 Pontos fortes e pontos fracos

Surgem agora questões importantes em relação aos *frameworks*, tais como desenvolvimento, documentação e uso. Todas elas apresentam seus pontos fortes e pontos fracos, comentados a seguir.

Desenvolvimento de *frameworks*

As metas são produzi-los de forma que possam ser genéricos e flexíveis. O fato de ser genérico surge pela generalização de um domínio de aplicação e um *framework* é flexível quando possui a capacidade de ser alterado (alterações na estrutura das classes e seus relacionamentos) e estendido (acréscimos de novas classes). Apresenta como ponto forte a reutilização de código e projeto com conseqüente aumento de produtividade e como ponto fraco está a complexidade [ROB98]. Silva [SIL00] também coloca que o desenvolvimento de um *framework* é complexo. Aborda ainda que as metodologias de desenvolvimentos existentes, em sua maioria, estão voltadas para produzir código e não utilizam uma notação de alto nível. Mesmo que se use os preceitos dos projetos e análises orientados a objetos (metodologias OOAD), elas não são adequadas a

frameworks. Mattsson [MAT99b] destaca os problemas que envolvem a composição de *frameworks* como uma forma de desenvolvimento de novos *frameworks*. Don Roberts e Ralph Johnson [ROB98] colocam que um *framework* deve ser desenvolvido apenas quando existirem várias aplicações sobre um domínio específico, de forma a tornar viável o tempo gasto no seu desenvolvimento.

Documentação de *frameworks*

A documentação de um *framework* deve contemplar três questões: o propósito do *framework*, como usá-lo e os detalhes do projeto do *framework*. Elas existem porque são destinadas a públicos diferentes. Enquanto existem pessoas que precisam decidir quais *frameworks* usar (neste caso uma breve descrição do propósito do *framework* é suficiente e as principais funcionalidades podem ser demonstradas através de exemplos), existem aquelas que precisam desenvolver implementações elementares (mais óbvias) a partir de um ou mais *frameworks*. Neste caso, faz-se necessário saber como usá-los, sem entrar em detalhes de projeto. Uma boa alternativa para saber usar um *framework* poderia ser através dos *cookbook* (livro de receitas). Por um outro lado, existem implementações não tão óbvias para as quais o conhecimento de detalhes do projeto de um *framework* é fundamental. Para estes casos, portanto, surge um terceiro grupo de pessoas envolvidas com a documentação de *frameworks:* aquelas que precisam saber detalhes de sua implementação, das classes abstratas ao modelo cooperativo de suas classes. A necessidade destes conhecimentos deve-se não apenas em procurar entender detalhes não documentados (ou até mesmo documentados) para a confecção de aplicações mais complexas, como também para modificar o *framework*.

Uma vez que os *frameworks* não requerem nenhuma tecnologia nova, não foi elaborado nenhum padrão específico para documentar todas suas especificações. E isto pode ser considerado como um ponto fraco da documentação de *frameworks*. As formas atuais mais comuns estão voltadas para a descrição textual, o próprio código do *framework* e o código das aplicações que os utilizam. Silva [SIL00] propõe o uso de UML com extensões sintáticas para adequar a notação às características específicas de

frameworks. Johnson [JOH92] propõe o uso de padrões para a documentação de *frameworks*.

Uso de *frameworks*

O uso de um *framework* não é o mesmo que o uso de uma biblioteca de classes. Uma aplicação baseada em um *framework* difere principalmente no controle que este exerce em tempo de execução no comportamento da aplicação, pois o fluxo de controle da aplicação está previamente definido nele e não é o desenvolvedor que o define. Já em aplicações que utilizam bibliotecas de classes, é tarefa do desenvolvedor definir as chamadas dos comportamentos das classes, determinando, assim, o fluxo de controle. A questão que surge é que a compreensão de um *framework* surge como um obstáculo a sua adoção. Assim como são difíceis de desenvolver, seu uso também depende muito da documentação apresentada pelo seu projetista.

Johnson e Foote [JOH88] classificam um *framework* quanto ao uso como "caixa branca" e "caixa preta". Uma aplicação que utiliza um *framework* como uma caixa branca adiciona métodos nas subclasses de uma ou mais classes. Apresenta alguns problemas, quando, por exemplo, o número de subclasses a serem criadas for relativamente grande e também quanto ao aprendizado de seu uso, uma vez que a adição destes novos métodos pelo mecanismo de herança exige o conhecimento da estrutura da classe. Uma forma de minimizar estes problemas surge com o *framework* caixa preta, onde seu uso dá-se pela disponibilização de um conjunto de subclasses concretas, reutilizáveis por composição como uma alternativa à reutilização por herança. Através da composição de classes, é possível usá-las sem o conhecimento profundo de suas funcionalidades.

Conforme já comentado, o ponto forte no uso de *frameworks* é o alto grau de reutilização de projeto e código, enquanto que os pontos fracos são as dificuldades para desenvolvê-los e usá-los.

B.2. Componentes

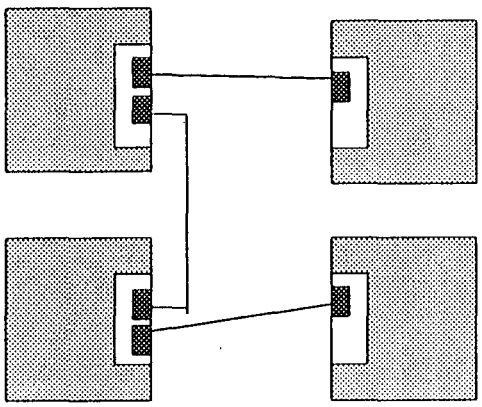
A idéia da confecção de software através da composição de componentes reutilizáveis é bastante anterior à programação orientada a objetos [SIL00, SZY99]. Em termos de especificação, um componente não necessariamente precisa ser escrito apenas com classes. Nas linguagens de programação que não provêem suporte à orientação a objetos, por exemplo, já são encontrados componentes em forma de função ou procedimento (uma função de uma biblioteca C é um componente).

Participantes do WCOP'96 definiram componente como "uma unidade de composição com interfaces contratualmente especificadas e dependências de contexto explícitas. Componentes podem ser desenvolvidos independentemente e estar sujeitos a composição com terceiros" [SZY96]. Uma interface especificada não significa ter que demonstrar como ela foi implementada, mas sim descrever um grupo de especificações de serviços do componente. O fato de serem independentes é garantido pelo encapsulamento de detalhes de implementação, o que permite que as funcionalidades sejam escritas em qualquer paradigma (orientadas a objetos, procedimental, *assembly*). Nem sempre teremos um componente para cada situação específica. Neste caso eles podem ser adaptados através da alteração do código fonte ou de *wrappers* [WEC97].

A figura B.3, extraída de Silva [SIL00], mostra um sistema composto a partir da conexão de vários componentes. Uma vez que uma interface é definida como uma coleção de pontos de acesso, em que cada um possui uma semântica especificada [WEC97], a forma de interação de um componente com o meio externo é através dos canais de comunicação.

Atualmente existem alguns padrões de especificação de componentes. Os mais conhecidos são CORBA, COM e JavaBeans. A OMG (*Object Management Group*), um consórcio de 700 empresas fundado em 1989, desenvolveu o CORBA (*Common Object Request Broker Architecture*) para permitir a interoperabilidade de objetos independentes de plataforma e localização física. Um objeto CORBA interage com

outro através de sua IDL (*Interface Definition Language*), puramente declarativa, sem conter detalhes de implementação. Atualmente estão disponibilizadas especificações IDL para as linguagens C, C++, Smalltalk e Java [ORF97, SZY99]. O segundo padrão, COM (*Component Object Model*), é da Microsoft. Os componentes ActiveX e OLE (*Object Linking and Embeding*) podem ser escritos em várias linguagens e nada mais são do que componentes COM distribuídos (DCOM), que, à semelhança do CORBA, também utilizam uma IDL [ORF97, SZY99]. E, finalmente, o JavaBeans, da Sun Microsystems, componentes desenvolvidos em Java também com independência de plataforma de execução e localização [SZY99].



O retângulos cinza são componentes. Neles, observam-se as interfaces representadas pelos retângulos brancos. Os retângulos em preto são os pontos de acesso (os canais de comunicação)

FIGURA B.3: ESQUEMA DE COMPONENTES

O aumento do número de aplicações orientadas a componentes deve-se a disponibilidade destes padrões de especificação. Em nome de uma maior reutilização de código e projeto, a tentativa de conexão de componentes considerados incompatíveis era inevitável. Dois componentes podem ser conectados se tiverem suas interfaces compatíveis [SIL00].

Assim como nos *frameworks*, os componentes também tem uma classificação quanto à reutilização. Eles são ditos caixa branca quando permitem a inspeção e a modificação do componente reutilizado e caixa preta quando não permitem. Um caso

híbrido é denominado de "caixa cinza", quando apenas uma parte da implementação do componente é aberto para inspeção e alteração. E, por fim, um componente é reutilizado ao estilo "caixa de vidro" (*glassbox*) quando permite a inspeção da implementação, mas não sua modificação.

B.2.1. Pontos fortes e pontos fracos

Componentes oferecem grandes vantagens na reutilização e facilidade de manutenção do software, tal qual os *frameworks*. Entretanto, o uso da abordagem de programação orientada por componentes esbarra em alguns detalhes. Nielsen e Elmstrom [NIE99] relatam seus problemas no uso de componentes binários, principalmente em questões como cultura e organização, documentação e gerenciamento destes componentes. Colocam ainda que uma das maiores queixas da sua equipe de programação é quanto à responsabilidade que estavam assumindo pela utilização de componentes de terceiros com documentação precária. Bosch [BOS97] coloca que componentes precisam ser adaptados antes de serem utilizados, pois dificilmente estarão aptos para uso imediato. Esta adaptação pode ser feita por composição, o que nem sempre é uma tarefa fácil devido às incompatibilidades entre os componentes compostos.

Pode-se concluir, então, que tanto os *frameworks* quanto os componentes apresentam vantagens e desvantagens e que o uso combinado das duas abordagens poderia aumentar suas capacidades de reutilização, assim como diminuir suas deficiências.

APÊNDICE C

DESEMPENHO DA INTERCEPTAÇÃO DE MENSAGENS DO *OP*MOP

Uma preocupação dos projetistas de MOP refere-se ao desempenho do mecanismo de interceptação de mensagens, uma vez que a interceptação do metanível tem relação direta com o comportamento em tempo de execução do sistema. Desta forma, a implementação deste mecanismo deve produzir pouca influência no desempenho do ciclo[14] de execução dos métodos reflexivos [KIC91].

Para efeito de análise do desempenho da interceptação de mensagens do *OP*MOP, foram elaboradas duas aplicações, chamadas de DemoSEM e DemoCOM. Ambas utilizam a classe C_NB, que possui o método processa utilizado para comparação (figura C.1). A DemoSEM, utiliza processa sem qualquer recurso reflexivo e, a outra, aplica o mecanismo de interceptação neste mesmo método. A figura C.2 apresenta as interfaces destas aplicações. No projeto DemoCOM existe ainda a classe C_MN, que é a classe do metanível controladora de C_NB (figura C.3).

```
unit uNB;

interface

Type C_NB = Class
              Procedure processa;
            End;

implementation

Procedure C_NB.processa;
Begin
      Espera(1);
End;

end.
```

FIGURA C.1: CLASSE C_NB DE EXEMPLO

[14] Entende-se por ciclo de execução de um método o cumprimento das etapas de invocação, processamento e retorno ao local de chamada do método.

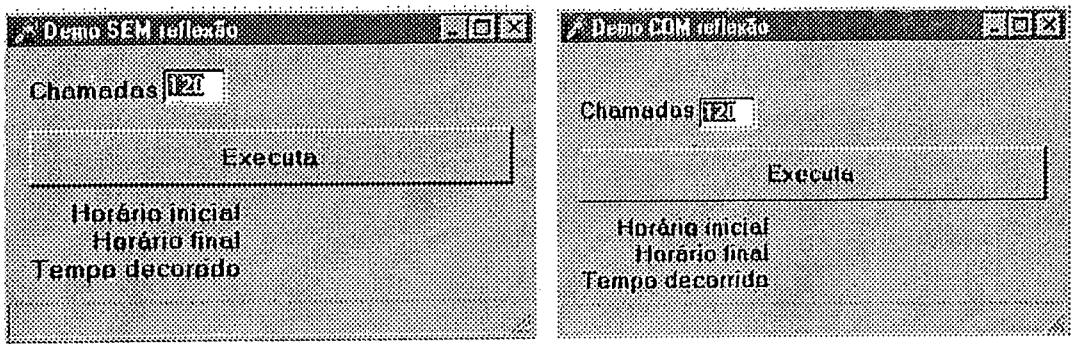


FIGURA C.2: INTERFACES DOS PROJETOS DEMOSEM E DEMOCOM

```
unit uMN;

interface

Uses
    Windows, Mensagens, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
    StdCtrls, uMOP;

Type
    C_MN = Class (TMOP)
        Public
            Function ControleExecucao (metodo: TMOP_Metodo; args: TMOP_Args):
                                                    Boolean; Override;
        End;

Implementation

Uses uNBControle;

Function C_MN.controleExecucao (metodo: TMOP_Metodo; args: TMOP_Args) : Boolean;
Begin
    Result := oNBControle.executa(metodo, args);
End;

end.
```

FIGURA C.3: CLASSE C_MN DE EXEMPLO

Os testes elaborados para a análise do desempenho da interceptação de mensagens possuem como idéia básica a comparação do tempo de execução entre um método reflexivo e um não reflexivo. A seguir, a descrição de cada teste.

C.1. Teste 1 – tempo de execução

A metodologia utilizada abrangeu a coleta dos dados e a apresentação dos resultados. As variáveis envolvidas são duas: "N", como o número de chamadas do método `processa`; e "Tempo Decorrido", como o tempo de execução das N interações do método.

C.1.1. - Coleta de Dados

Os dados coletados foram os tempos decorridos provenientes da execução dos projetos `DemoSEM` e `DemoCOM`.

Basicamente, cada projeto executou as seguintes etapas:

- Obtenção do horário atual antes das invocações de `processa`
- Invocação do método `processa`, da classe `C_NB`, N vezes. Para o projeto `DemoSEM`, `processa` é não reflexivo e, para `DemoCOM`, é reflexivo
- Obtenção do horário atual após esta interação.
- Cálculo do tempo decorrido, em minutos, das N invocações.

C.1.2. – Apresentação dos resultados

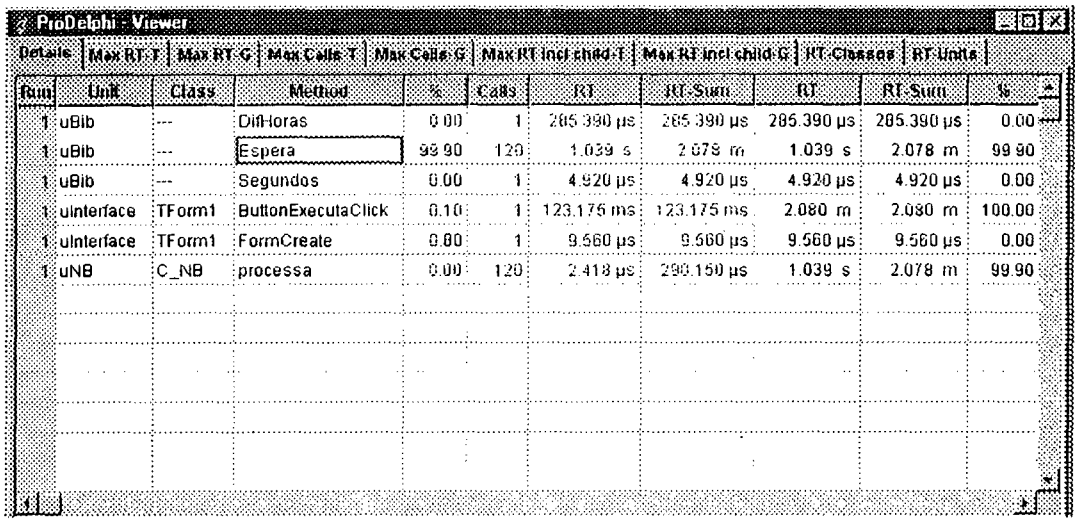
Os valores fornecidos pelas execuções dos projetos foram colocados numa planilha de eletrônica, conforme tabela C.4.

TABELA C.4: RESULTADOS DO TESTE 1

	Tempo Decorrido	
N	Reflexivo (min)	Não reflexivo (min)
120	2,100	2,083
300	5,233	5,217
600	10,450	10,433
1200	20,883	20,883

C.2. Teste 2 – uso do ProDelphi

Neste segundo teste da análise do desempenho do *OP*MOP, foi utilizado um software específico, denominado de ProDelphi. Basicamente, a função deste programa é medir o tempo de execução de subalgoritmos (procedimentos e funções), através da inserção de controles no início e no final dos blocos, em todos os arquivos do projeto. O projeto é, então, recompilado e, após sua execução, vários arquivos são criados pelo ProDelphi [ADO00]. Os resultados são apresentados através dos seus arquivos específicos ou através de sua interface (figura C.5).



Num	Unit	Class	Method	%	Calls	RT	RT-Sum	RT	RT-Sum	%
1	uBib	---	DifHoras	0.00	1	285.390 µs	285.390 µs	285.390 µs	285.390 µs	0.00
1	uBib	---	Espera	99.90	120	1.039 s	2.078 m	1.039 s	2.078 m	99.90
1	uBib	---	Segundos	0.00	1	4.920 µs	4.920 µs	4.920 µs	4.920 µs	0.00
1	uInterface	TForm1	ButtonExecutaClick	0.10	1	123.175 ms	123.175 ms	2.080 m	2.080 m	100.00
1	uInterface	TForm1	FormCreate	0.00	1	9.560 µs	9.560 µs	9.560 µs	9.560 µs	0.00
1	uNB	C_NB	processa	0.00	120	2.418 µs	290.150 µs	1.039 s	2.078 m	99.90

As colunas mais relevantes são:
Calls: número de invocações do método
RT: tempo total de chamadas do método sem considerar seus procedimentos filhos
RT-Sum: Calls * RT
RT: tempo total de chamadas do método considerando seus procedimentos filhos
RT-Sum: Calls * RT

FIGURA C.5: INTERFACE DO PRODELPHI

C.2.1. - Coleta de Dados

Os dados coletados foram os tempos decorridos provenientes da execução dos projetos DemoSEM e DemoCOM, que estavam com as devidas adaptações feitas por ProDelphi.

C.2.2. – Apresentação dos resultados

Os valores fornecidos pelas execuções dos projetos foram colocados numa planilha de eletrônica, conforme tabela C.6.

TABELA C.6: RESULTADOS DO TESTE 2

	Tempo Decorrido			
	Reflexivo		Não reflexivo	
Calls	RT (s)	RT-Sum (min)	RT (s)	RT-Sum (min)
120	1,044	2,088	1,039	2,078
300	1,041	5,204	1,039	5,195
600	1,039	10,394	1,039	10,394
1200	1,039	20,787	1,039	20,787

Analisando-se os resultados dos dois testes, chega-se a algumas conclusões:

- No teste 1, os tempos provenientes das execuções dos métodos reflexivos e não reflexivos foram praticamente os mesmos. Uma vez que o *overhead* é fixo, as diferenças entre as invocações pode ser atribuída ao mecanismo de interceptação de mensagens promovido por *OP*MOP;
- No teste 2, quanto maior o número de chamadas, mais os valores das colunas RT-Sum das execuções dos métodos reflexivos e não reflexivos aproximam-se;
- As diferenças no formato de medição entre os testes 1 e 2 ficam evidenciadas pelos valores apurados, conforme tabelas C.4 e C.6. Enquanto que no teste 1 o tempo decorrido foi obtido considerando-se não só a invocação do método, mas também a estrutura de repetição responsável pelas várias interações, no teste 2, os programas fontes dos projetos foram alterados pelo ProDelphi, o qual colocou um código no início e final do algoritmo das funções e dos procedimentos.

De um modo geral, estes testes comparativos serviram para mostrar que o desempenho da execução do *OP*MOP não deve ser comprometedor. Entretanto, os números resultantes não devem ser considerados como finais. Uma análise mais profunda faz-se necessária para a obtenção de resultados mais precisos e confiáveis.

APÊNDICE D

EXEMPLOS DETALHADOS DE USO DOS COMPONENTES *OP*MOP

Esta parte do trabalho mostra, em detalhes, os exemplos apresentados no capítulo da implementação. Em alguns deles, são destacadas versões reflexivas e não reflexivas de um mesmo exemplo. Apenas para relembrar, os nomes de *units* e classes foram padronizados, conforme a seguir:

(a) o projeto: será sempre `Project1`.

(b) a classe base: está na *unit* `uNivelBase` e tem o nome de `TC_NivelBase`;

(c) a interface: está numa *unit* `uInterface` e contém um *form* chamado `Form1`;

(d) a metaclasse: está em `uMetaNivel` e tem o nome de `TC_MetaNivel`;

(e) os componentes não visuais: estão na *unit* `uDataModule`, que contém um *DataModule* chamado `DM`;

(f) os componentes visuais: estão no próprio `Form1`.

D.1. Aplicação 1 – Depurador

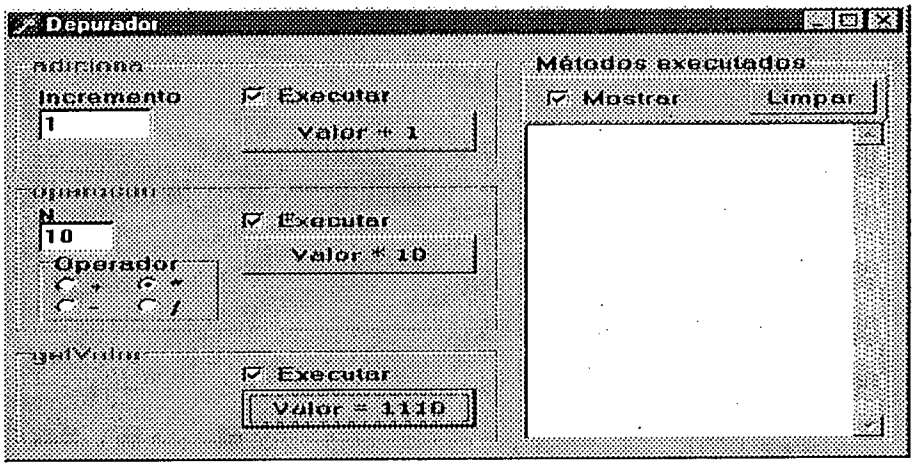
O exemplo atua como um depurador, permitindo que o usuário, em tempo de execução, controle a invocação e a apresentação dos métodos.

```
program Project1;

uses
  Forms,
  uInterface in 'uInterface.pas' {Form1},
  uNivelBase in 'uNivelBase.pas',
  uMetaNivel in 'uMetaNivel.pas',
  uDataModule in 'uDataModule.pas' {DM: TDataModule};

{$R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TDM, DM);
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

(a) o projeto

```
unit uNivelBase;

Interface

Type TC_NivelBase = Class
     Private
          valor : Integer;
     Public
          Constructor Init;
          Procedure   adiciona (i : Integer);
          Function    getValor : Integer;
          Procedure   operacao (n : Byte; op : String);
     End;

Implementation

Constructor TC_NivelBase.Init;                    Begin valor := 0;          End;
Procedure   TC_NivelBase.adiciona (i : Integer); Begin valor := valor + i; End;
Function    TC_NivelBase.getValor : Integer;      Begin getValor := valor;  End;
Procedure   TC_NivelBase.operacao (n : Byte; op : String);
Begin
     If op = '+' Then valor := valor + n;
     If op = '-' Then valor := valor - n;
     If op = '*' Then valor := valor * n;
     If op = '/' Then valor := valor Div n;
End;

end.
```

(b) a classe base



(c) o *form* interface

Aparência final da interface, aplicação não reflexiva ainda, após três cliques nos botões "Valor + 1" e "Valor * 10". Notar que os métodos executados não foram mostrados.

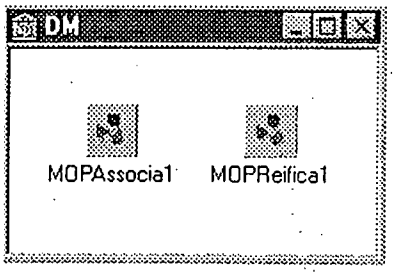
```
unit uInterface;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
→  uNivelBase;

type TForm1 = class(TForm)
              GroupBox1: TGroupBox;
              Label2: TLabel;
              ...
              Memo1: TMemo;
              Button1: TButton;
              procedure FormCreate(Sender: TObject);
              procedure ButtonGetValorClick(Sender: TObject);
              procedure ButtonAdicionaClick(Sender: TObject);
              procedure ButtonOperacaoClick(Sender: TObject);
     private
→         MinhaNB : TC_NivelBase;
     end;

var Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
→    MinhaNB := TC_NivelBase.Create;
     MinhaNB.init;
end;

procedure TForm1.ButtonAdicionaClick(Sender: TObject);
begin
     MinhaNB.adiciona (StrToInt(EditAdiciona.Text));
end;

procedure TForm1.ButtonOperacaoClick(Sender: TObject);
begin
     MinhaNB.operacao (StrToInt(EditN.Text),strOp);
end;

procedure TForm1.ButtonGetValorClick(Sender: TObject);
begin
     ButtonGetValor.Caption := 'Valor = ' + IntToStr(MinhaNB.getValor);
end;

end.
```

(c) a *unit* interface

A *unit* da interface do *form* anterior. Em destaque, as referências à classe base. As linhas marcadas com → indicam que a aplicação não está reflexiva, pois a classe instanciada não foi alterada.

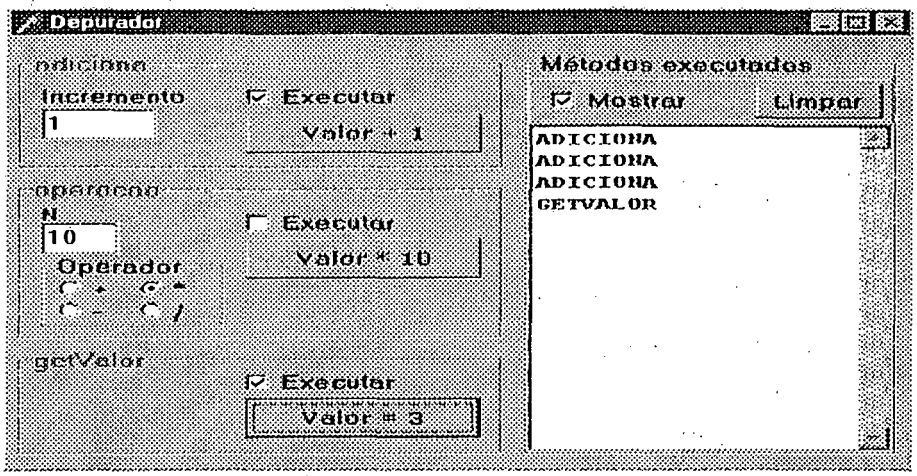
```
unit uMetaNivel;

interface

Uses Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
     StdCtrls, uMOP;

Type
     TC_MetaNivel = Class (TMOP)
       Public
           function controleExecucao (metodo: TMOP_Metodo; args: TMOP_Args):
                                                          Boolean; Override;
     End;

implementation

Uses uInterface, uNivelBaseControle;

Function TC_MetaNivel.controleExecucao (metodo: TMOP_Metodo; args: TMOP_Args):
                                                                  Boolean;
Begin
     Result := False;
     metodo.nome := Trim(UpperCase(metodo.nome));

     If (metodo.nome = 'ADICIONA') And (Form1.CBAdiciona.Checked) Then Begin
        If Form1.CBMostrar.Checked Then Form1.Memo1.Lines.Append(metodo.nome);
        Result := oNBControle.executa(metodo, args);
     End;

     If (metodo.nome = 'OPERACAO') And (Form1.CBOperacao.Checked) Then Begin
        If Form1.CBMostrar.Checked Then Form1.Memo1.Lines.Append(metodo.nome);
        Result := oNBControle.executa(metodo, args);
     End;

     If (metodo.nome = 'GETVALOR') And (Form1.CBGetValor.Checked) Then Begin
        If Form1.CBMostrar.Checked Then Form1.Memo1.Lines.Append(metodo.nome);
        Result := oNBControle.executa(metodo, args);
     End;
End;

end.
```

(d) a metaclasse




```
unit uDataModule;

interface

uses Windows, Messages, OPMOP;

type TDM = class(TDataModule)
             MOPAssocia1: TMOPAssocia;
             MOPReifica1: TMOPReifica;
           end;

var DM: TDM;

implementation

{$R *.DFM}

end.
```

(e) o *form* e a *unit* do *datamodule*



(c) o *form* interface

Aparência final da interface, estando a aplicação reflexiva, após três cliques nos botões "Valor + 1" e "Valor * 10". Notar que o método Operacao não apareceu junto a lista dos executados porque, de fato, não foi executado (sua opção Executar não está marcada, como nos demais métodos).

```
      unit uInterface;

      interface

      uses Windows, Messages, SysUtils, Classes, Graphics, Controls,
→          (uNivelBase)uNivelBaseControle;

      type TForm1 = class(TForm)
                GroupBox1: TGroupBox;
                Label2: TLabel;
                 ...
          private
→               MinhaNB : (TC_NivelBase) TC_NivelBaseAlterada;
          end;

      var Form1: TForm1;

      implementation
      {$R *.DFM}

      procedure TForm1.FormCreate(Sender: TObject);
      begin
→         MinhaNB := (TC_NivelBase)TC_NivelBaseAlterada.Create;
          MinhaNB.init;
      end;

      ...

      end.
```

(c) a *unit* interface

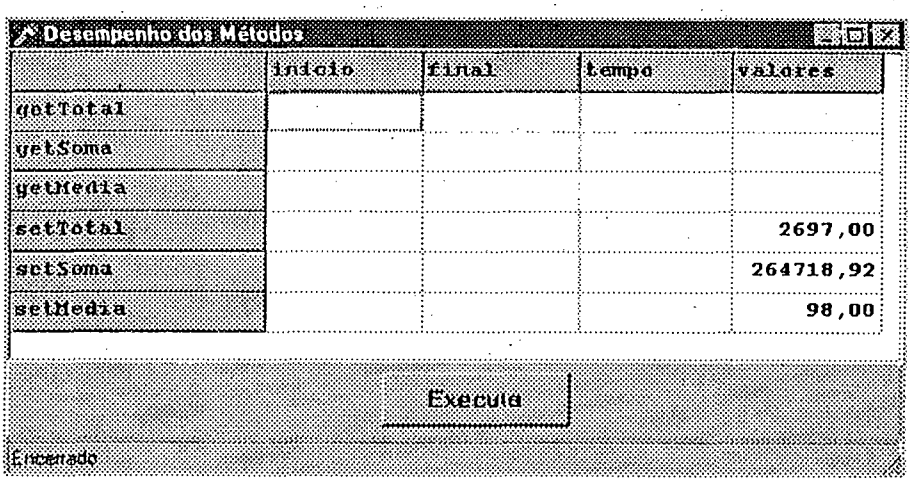
A *unit* da interface do *form* anterior. As linhas marcadas com → indicam que a aplicação já sofreu a ação do componente TMOPAssocia (prover as adaptações necessárias ao processo de interceptação de mensagens). Neste ponto, então, a aplicação está reflexiva.

D.2. Aplicação 2 – Desempenho

Outro exemplo relacionado à interceptação de mensagens, com o intuito de medir o desempenho de invocações de métodos. A função da metaclasse, aqui, é interceptar as mensagens, obter os horários inicial e final de cada invocação e calcular o tempo decorrido.

```
program Project1;

uses forms,
     uInterface in 'uInterface.pas' {Form1},
     uNivelBase in 'uNivelBase.pas',
     uMetaNivel in 'uMetaNivel.pas',
     uDataModule in 'uDataModule.pas' {DM: TDataModule};

{$R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TDM, DM);
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

(a) o projeto



(c) o *form* da interface

Como a aplicação não está reflexiva, apenas os resultados da aplicação foram mostrados. A estatística de desempenho, que é tratada pela metaclasse, não foi feita.

```
unit uInterface;

interface

uses Windows, Messages, SysUtils, Classes, Graphics, Controls,
→     uNivelBase;

type TForm1 = class(TForm)
            ...
            ButtonExecuta: TButton;
            procedure FormCreate(Sender: TObject);
            procedure ButtonExecutaClick(Sender: TObject);
    private
→           minhaNB : TC_NivelBase;
            Procedure PreencheGrid;
    end;

var Form1: TForm1;

implementation
{$R *.DFM}
Uses uDataModule, uMOP;

procedure TForm1.FormCreate(Sender: TObject);
begin
→     minhaNB := TC_NivelBase.Create;
      PreencheGrid;
end;

Procedure TForm1.PreencheGrid;
Var x,    : Integer;
    pMet  : TP_Metodos;
Begin
      ...
      // preencher nomes das linhas do grid (nomes dos métodos da classe)
      SG.RowCount := DM.MOPReifical.Classe.Metodos.NroItens + 1;
      New(pMet);
      For x := 0 To DM.MOPReifical.Classe.Metodos.NroItens -1 Do Begin
          pMet := DM.MOPReifical.Classe.Metodos.Metodos.Items[x];
          SG.Cells[0,x+1] := pMet^.nome;
      End;
End;

procedure TForm1.ButtonExecutaClick(Sender: TObject);
begin
      // inicializações
      PreencheGrid;

      // invoca os métodos TOTAL e mostra o resultado
      minhaNB.setTotal ('COUNT(ValorEmiDoc)',DM.Table1.TableName);
      SG.Cells[4,4] := Format('%10.2f', [minhaNB.getTotal]);

      // invoca os métodos SOMA e mostra o resultado
      minhaNB.setSoma ('SUM(ValorEmiDoc)',DM.Table1.TableName);
      SG.Cells[4,5] := Format('%10.2f', [minhaNB.getSoma]);

      // invoca os métodos MEDIA e mostra o resultado
      minhaNB.setMedia ('AVG(ValorEmiDoc)',DM.Table1.TableName);
      SG.Cells[4,6] := Format('%10.2f', [minhaNB.getMedia]);
end;

end.
```

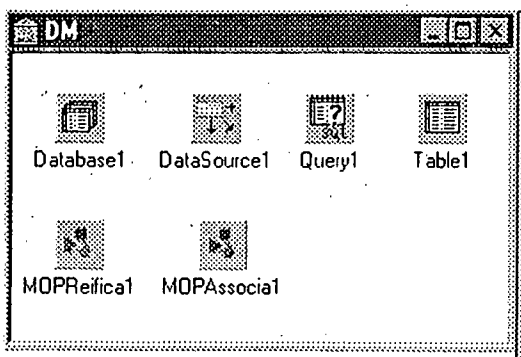
(c) a *unit* do último *form* apresentado

As linhas em destaque são referências à classe base. As indicadas com → indicam que a aplicação não está reflexiva, porque não foram feitas as adaptações à interceptação de mensagens.

```
unit uNivelBase;
interface

Type TC_NivelBase = Class
     Private
          total : Real;
          soma  : Real;
          media : Real;
     Public
          Function  getTotal : Real;
          Function  getSoma  : Real;
          Function  getMedia : Real;
          Procedure setTotal (argCampo, argTabela : String);
          Procedure setSoma  (argCampo, argTabela : String);
          Procedure setMedia (argCampo, argTabela : String);
     End;

Implementation
Uses uDataModule;

Function executaSQL (argCampo, argTabela : String) : String;
Var i,f : Byte;
Begin
     DM.Query1.Close;
     DM.Query1.SQL.Clear;
     DM.Query1.SQL.Add('select '+argCampo+' from '+argTabela);
     DM.Query1.ExecSQL;
     DM.Query1.Open;

     i := Pos('(',argCampo);
     f := Pos(')',argCampo);
     If (i <> 0) And (f <> 0) Then executaSQL := Copy(argCampo,i+1,f-i-1)
                              Else executaSQL := '';
End;

Function TC_NivelBase.getTotal : Real;  Begin  getTotal := total;  End;
Function TC_NivelBase.getSoma  : Real;  Begin  getSoma  := soma;   End;
Function TC_NivelBase.getMedia : Real;  Begin  getMedia := media;  End;

Procedure TC_NivelBase.setTotal (argCampo, argTabela : String);
Var coluna : String;
Begin
     coluna := executaSQL (argCampo, argTabela);
     DM.Query1.First;
     total := DM.Query1.FieldByName('COUNT of '+coluna).AsFloat;
End;

Procedure TC_NivelBase.setSoma (argCampo, argTabela : String);
Var coluna : String;
Begin
     coluna := executaSQL (argCampo, argTabela);
     DM.Query1.First;
     soma := DM.Query1.FieldByName('SUM of '+coluna).AsFloat;
End;

Procedure TC_NivelBase.setMedia (argCampo, argTabela : String);
Var coluna : String;
Begin
     coluna := executaSQL (argCampo, argTabela);
     DM.Query1.First;
     media := DM.Query1.FieldByName('AVERAGE of '+coluna).AsInteger;
End;

end.
```

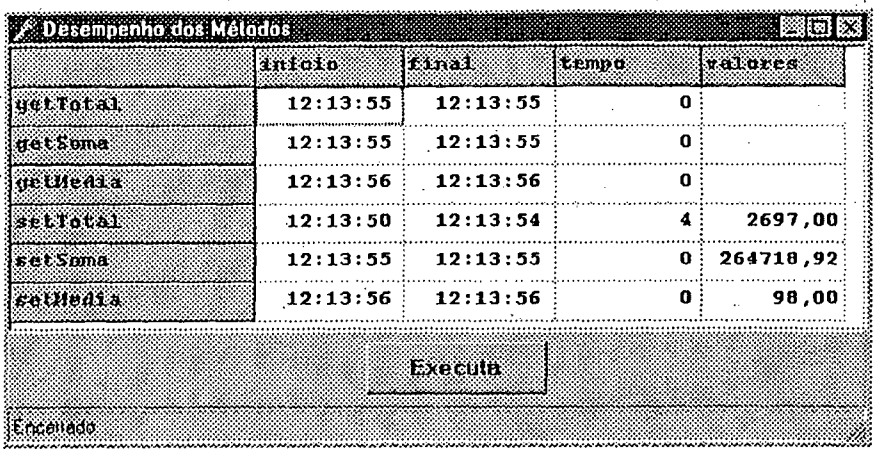
```
unit uMetaNivel;

interface
Uses Windows, Messages, SysUtils, Classes, Graphics, Controls, StdCtrls, uMOP;

Type TC_MetaNivel = Class (TMOP)
            Public
               Procedure interceptaAntes  (metodo: TMOP_Metodo); Override;
               Procedure interceptaDepois (metodo: TMOP_Metodo); Override;
               Function  controleExecucao (metodo: TMOP_Metodo; args: TMOP_Args) :
                                                    Boolean; Override;
            End;
implementation
Uses uBib, uInterface, uNivelBaseControle;
Var  v : Array[1..50] Of Record
                           Inicio, final, tempo : String;
                         End;

Procedure TC_MetaNivel.interceptaAntes (metodo: TMOP_Metodo);
Begin
     // obter o horário atual e mostrá-lo
     v[metodo.id].inicio := TimeToStr(Now);
     Form1.SG.Cells[1,metodo.id] := Format('%10s',[v[metodo.id].inicio]);
End;

Procedure TC_MetaNivel.interceptaDepois (metodo: TMOP_Metodo);
Var difHora : String;
    difSeg  : Integer;
Begin
     // obter o horário atual e mostrá-lo
     v[metodo.id].final := TimeToStr(Now);
     Form1.SG.Cells[2,metodo.id] := Format('%10s',[v[metodo.id].final]);

     // calcular a diferença de horários e mostrá-la
     difHora := DifHoras(v[metodo.id].inicio, v[metodo.id].final);
     difSeg  := Segundos(StrToTime(difHora));
     v[metodo.id].tempo := IntToStr(difSeg);
     Form1.SG.Cells[3,metodo.id] := Format('%10s',[v[metodo.id].tempo]);
End;

Function TC_MetaNivel.controleExecucao (metodo: TMOP_Metodo; args: TMOP_Args) :
                                                              Boolean;
Begin
     Result := oNBControle.executa(metodo, args);
End;

end.
```

(d) a metaclasse




```
unit uDataModule;
interface
uses Windows, Messages, DBTables, OPMOP;

Type  TDM = class(TDataModule)
            DataSource1: TDataSource;
            Table1: TTable;
            Query1: TQuery;
            Database1: TDatabase;
            MOPReifica1: TMOPReifica;
            MOPAssocia1: TMOPAssocia;
        End;
Var DM: TDM;

implementation
{$R *.DFM}
end.
```

(f) o *form* e a *unit* do *datamodule*



(c) o *form* da interface

Neste ponto, a aplicação já está reflexiva. A tarefa da metaclasse é, a cada invocação, obter o horário inicial e final e calcular o tempo decorrido dos métodos da classe base. As colunas preenchidas demonstram isto.

```
unit uInterface;

interface

uses Windows, Messages, SysUtils, Classes, Graphics, Controls,
→       {uNivelBase}uNivelBaseControle;

type TForm1 = class(TForm)

            ButtonExecuta: TButton;
            procedure FormCreate(Sender: TObject);
            procedure ButtonExecutaClick(Sender: TObject);
    private
→           minhaNB : {TC_NivelBase}TC_NivelBaseAlterada;
            Procedure PreencheGrid;
    end;

var Form1: TForm1;

implementation
{$R *.DFM}
Uses uDataModule, uMOP;

procedure TForm1.FormCreate(Sender: TObject);
begin
→       minhaNB := {TC_NivelBase}TC_NivelBaseAlterada.Create;
        PreencheGrid;
end;

...

end.
```

(c) o *unit* do *form* anterior

As adaptações provocadas pela atuação de TMOPAssocia estão nas linhas indicadas com →. O restante do programa fonte não foi alterado.

D.3. Aplicação 3 – Meta-informações em *designtime*

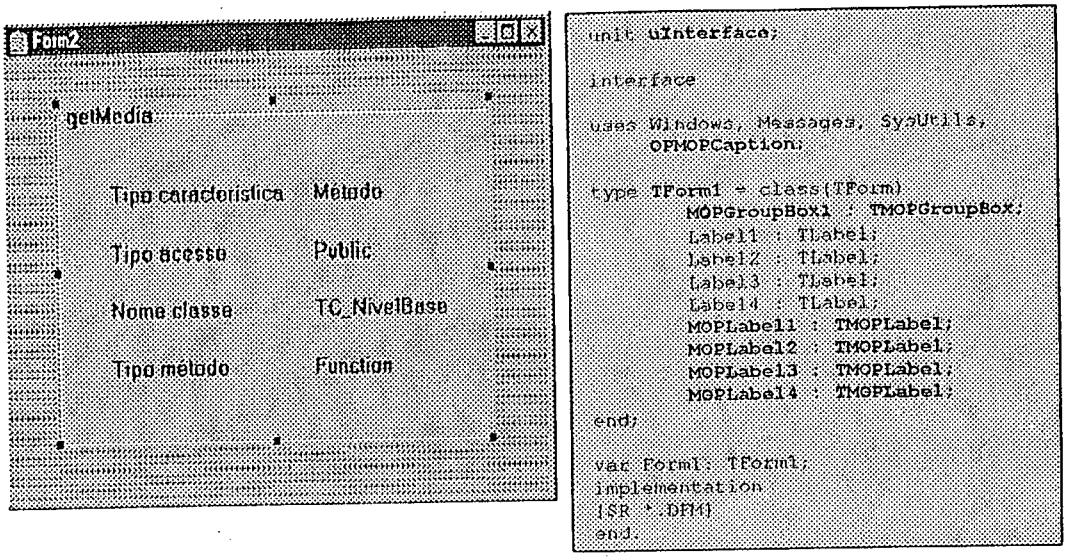
Este exemplo não tem nenhuma metaclasse definida. Isto porque o resultado da reificação, promovida por TMOPReifica, apenas apresenta as meta-informações da classe base através dos componentes visuais *OP*MOP.

```
program Project1;

uses Forms,
     uInterface in 'uInterface.pas' {Form1},
     uNivelBase in 'uNivelBase.pas',
     uDataModule in 'uDataModule.pas' {DM: TDataModule};

{$R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.CreateForm(TDM, DM);
  Application.Run;
end.
```

(a) o projeto

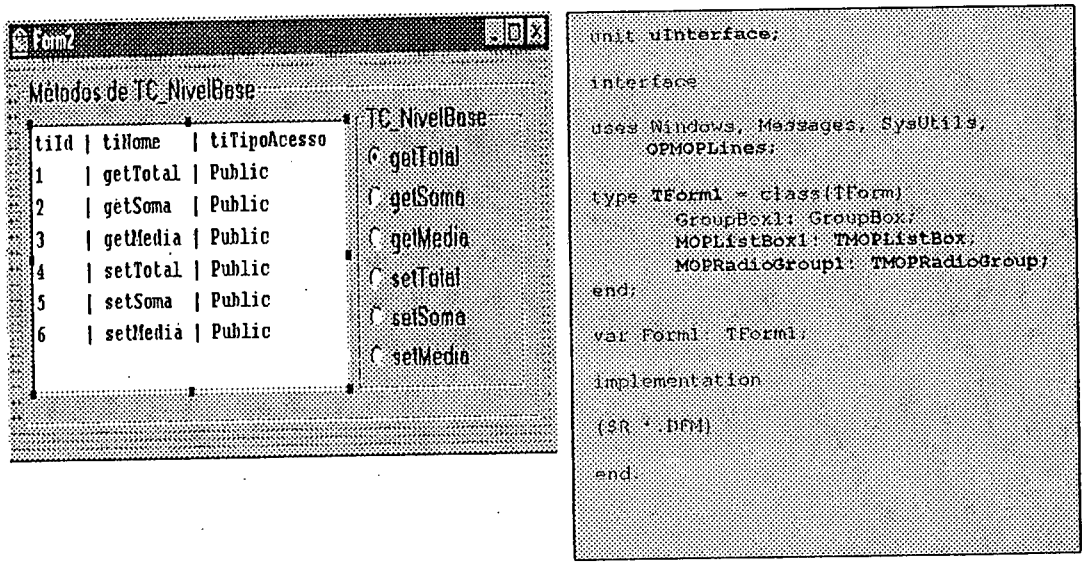
```
unit uNivelBase;
interface

Type TC_NivelBase = Class
     Private
          total : Real;
          soma  : Real;
          media : Real;
     Public
          Function  getTotal : Real;
          Function  getSoma  : Real;
          Function  getMedia : Real;
          Procedure setTotal (argCampo, argTabela : String);
          Procedure setSoma  (argCampo, argTabela : String);
          Procedure setMedia (argCampo, argTabela : String);
     End;

implementation

Function  TC_NivelBase.getTotal : Real;                                  Begin End;
Function  TC_NivelBase.getSoma  : Real;                                  Begin End;
Function  TC_NivelBase.getMedia : Real;                                  Begin End;
Procedure TC_NivelBase.setTotal (argCampo, argTabela : String);  Begin End;
Procedure TC_NivelBase.setSoma  (argCampo, argTabela : String);  Begin End;
Procedure TC_NivelBase.setMedia (argCampo, argTabela : String);  Begin End;

end.
```

(b) a classe base



(c) o *form* e a *unit*

Em destaque, os componentes do grupo *string*, presentes em OPMOPCaption, utilizados no *form*.



(c) o *form* e a *unit*

. Em destaque, os componentes do grupo *lines*, presentes em OPMOPLines, utilizados no *form*.

D.4. Aplicação 4 – Meta-informações em *runtime*

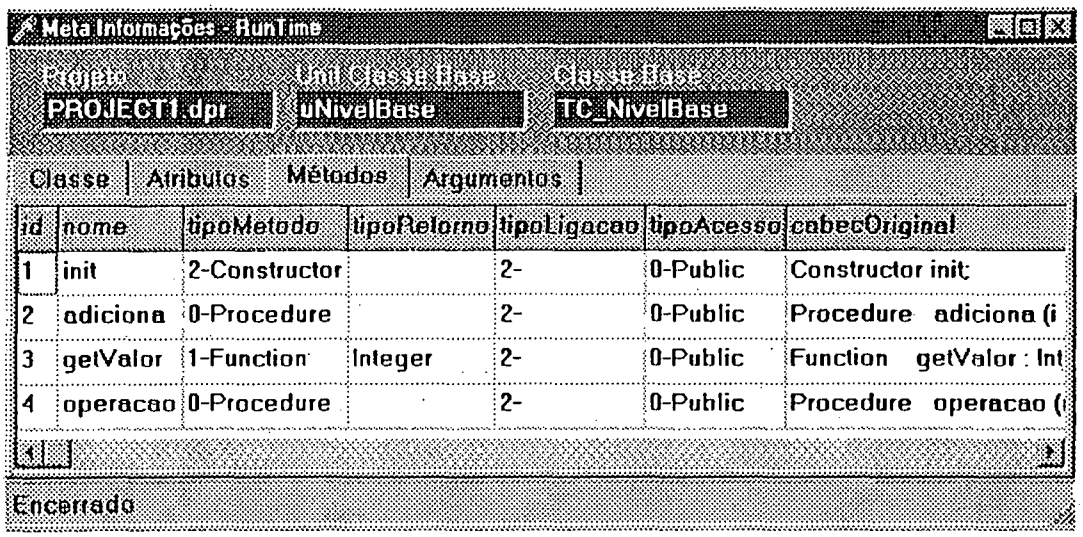
Ao contrário do exemplo anterior, onde o resultado da reificação é mostrado em tempo de projeto, este, da aplicação 4, mostra como as meta-informações da classe base são obtidas em tempo de execução.

```
program Project1;

uses Forms,
     uInterface in 'uInterface.pas' {Form1},
     uNivelBase in 'uNivelBase.pas';

{$R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

(a) o projeto

```
unit uNivelBase;

interface

Type TC_NivelBase = Class
     Private
          valor : Integer;
     Public
          Constructor init;
          Procedure   adiciona (i : Integer);
          Function    getValor : Integer;
          Procedure   operacao (n : Byte; op : String);
     End;
implementation

Constructor TC_NivelBase.init;                  Begin valor  := 0;          End;
Procedure   TC_NivelBase.adiciona (i : Integer); Begin valor := valor + i; End;
Function    TC_NivelBase.getValor : Integer;    Begin getValor := valor;    End;
Procedure   TC_NivelBase.operacao (n : Byte; op : String);
Begin
     If op = '+' Then valor := valor +  n;
     If op = '-' Then valor := valor -  n;
     If op = '*' Then valor := valor *  n;
     If op = '/' Then valor := valor Div n;
End;

end.
```

(b) a classe base



(c) o *form* da interface

```
unit uInterface;

interface

uses Windows, Messages, SysUtils, Classes, Graphics, OPMOP;

type TForm1 = class(TForm)
    ...
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    MOPReifica1 : TMOPReifica;
    Procedure Reificacao;
    Procedure PreencherGrids;
  end;

var Form1: TForm1;
implementation
{$R *.DFM}

Uses uMOP;

procedure TForm1.FormCreate(Sender: TObject);
begin
    MOPReifica1 := TMOPReifica.Create(Self);
    Reificacao;
    PreencherGrids;
End;

procedure TForm1.FormDestroy(Sender: TObject);
begin
    MOPReifica1.Destroy;
end;

Procedure TForm1.Reificacao;
Var nomeProj, extProj : String;
Begin
    // obter informações do projeto
    nomeProj := ExtractFileName(Application.ExeName);
    extProj  := ExtractFileExt(Application.ExeName);
    Delete(nomeProj,Pos(extProj,nomeProj),Length(extProj));
    nomeProj := nomeProj + '.dpr';

    // promove a reificação
    MOPReifica1.Projeto         := nomeProj;
    MOPReifica1.NivelBaseUnit   := 'uNivelBase';
    MOPReifica1.NivelBaseClasse := 'TC_NivelBase';
    MOPReifica1.Reifica         := True;

    // mostra no cabeçalho do form
    EProjeto.Text  := MOPReifica1.Projeto;
    EUCB.Text      := MOPReifica1.NivelBaseUnit;
    ECB.Text       := MOPReifica1.NivelBaseClasse;
End;
```

<continua na página seguinte>

```
        <continuação da página anterior>

Procedure TForm1.PreencherGrids;
Var c, i : Byte;
    pAtrib : TP_Atributos;
    pMet   : TP_Metodos;
    pArgs  : TP_Args;
Begin
    // Abandona, se não tem classe reificada
    If MOPReifical.Classe.Nome = '' Then Exit;

    // Classe
    SG1.Cells[0,1] := MOPReifical.Classe.Nome;
    SG1.Cells[1,1] := MOPReifical.Classe.NomePai;

    // Atributos
    New(pAtrib);
    SG2.RowCount := 1 + MOPReifical.Classe.Atributos.NroItens;
    For c := 0 To (MOPReifical.Classe.Atributos.NroItens - 1) Do Begin
        pAtrib := MOPReifical.Classe.Atributos.Atributos.Items[c];
        SG2.Cells[0,c+1] :=          IntToStr(pAtrib^.id);
        SG2.Cells[1,c+1] :=                   pAtrib^.nome;
        SG2.Cells[2,c+1] :=                   pAtrib^.tipoDado;
        SG2.Cells[3,c+1] :=    IntToStr(Ord(pAtrib^.tipoAcesso)) + '-' +
                          TextoTipoAcesso(pAtrib^.tipoAcesso);
    End;

    // Metodos
    New(pMet);
    SG3.RowCount := 1 + MOPReifical.Classe.Metodos.NroItens;
    For c := 0 To (MOPReifical.Classe.Metodos.NroItens - 1) Do Begin
        pMet := MOPReifical.Classe.Metodos.Metodos.Items[c];
        SG3.Cells[0,c+1] :=          IntToStr(pMet^.id);
        SG3.Cells[1,c+1] :=                   pMet^.nome;
        SG3.Cells[2,c+1] :=    IntToStr(Ord(pMet^.tipoMetodo))+'-'+
                          TextoTipoMetodo(pMet^.tipoMetodo);
        SG3.Cells[3,c+1] :=                   pMet^.tipoRetorno;
        SG3.Cells[4,c+1] :=    IntToStr(Ord(pMet^.tipoLigacao))+'-'+
                          TextoTipoLigacao(pMet^.tipoLigacao);
        SG3.Cells[5,c+1] :=    IntToStr(Ord(pMet^.tipoAcesso)) +'-'+
                          TextoTipoAcesso(pMet^.tipoAcesso);
        SG3.Cells[6,c+1] :=                   pMet^.cabecOriginal;
    End;

    // Argumentos
    New(pArgs);
    SG4.RowCount := 1 + MOPReifical.Classe.Args.NroItens;
    For c := 0 To (MOPReifical.Classe.Args.NroItens - 1) Do Begin
        pArgs := MOPReifical.Classe.Args.Args.Items[c];
        SG4.Cells[0,c+1] :=          IntToStr(pArgs^.idMetodo);
        SG4.Cells[1,c+1] :=          IntToStr(pArgs^.id);
        SG4.Cells[2,c+1] :=                   pArgs^.nome;
        SG4.Cells[3,c+1] :=                   pArgs^.tipoDado;
        SG4.Cells[4,c+1] :=    IntToStr(Ord(pArgs^.tipoPassagem))+'-'+
                          TextoTipoPassagem(pArgs^.tipoPassagem);
        SG4.Cells[5,c+1] :=                   pArgs^.argsOriginal;
    End;
End;
```

(c) a *unit* do *form* anterior

Em destaque, a utilização do componente MOPReifica, criado em tempo de execução.